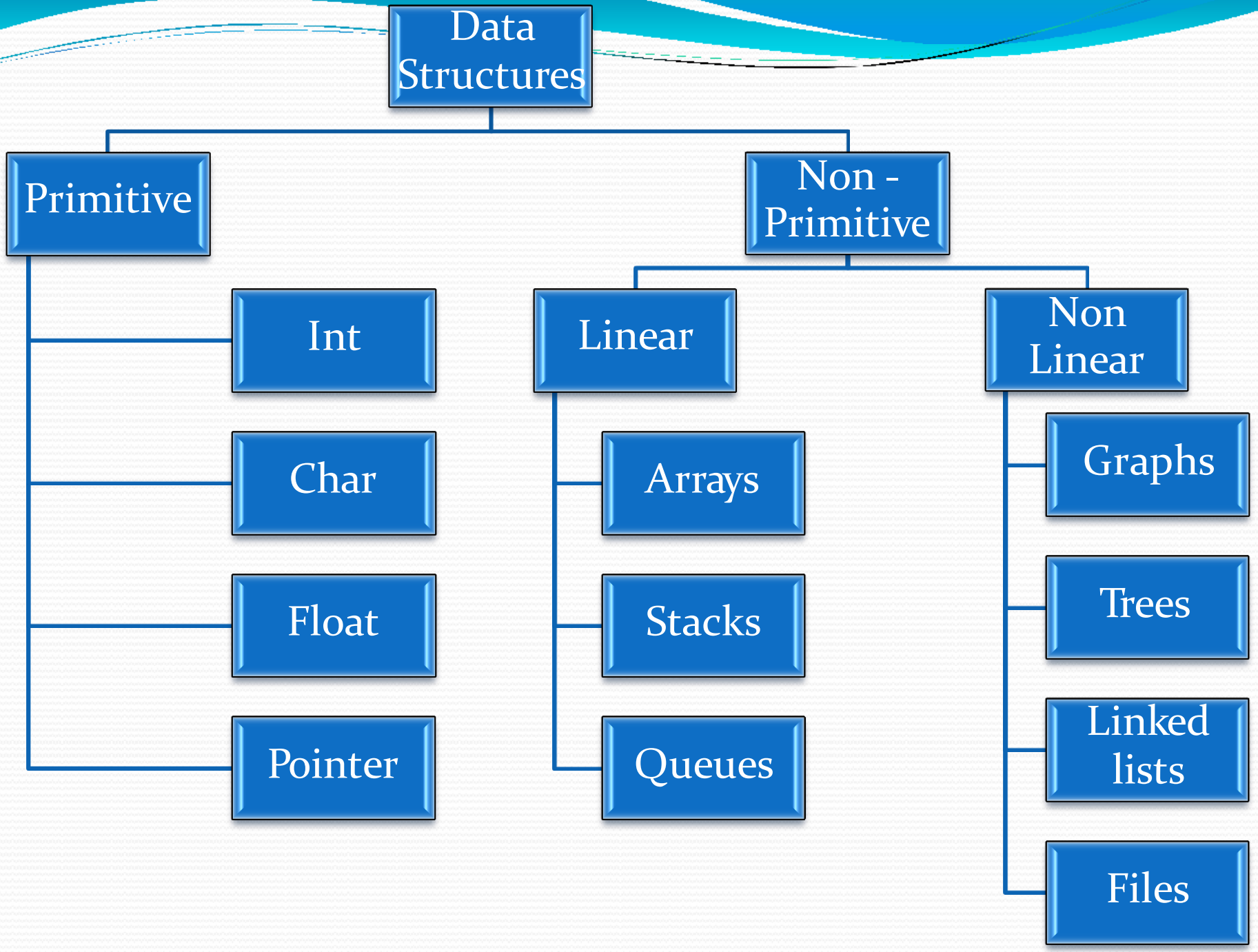
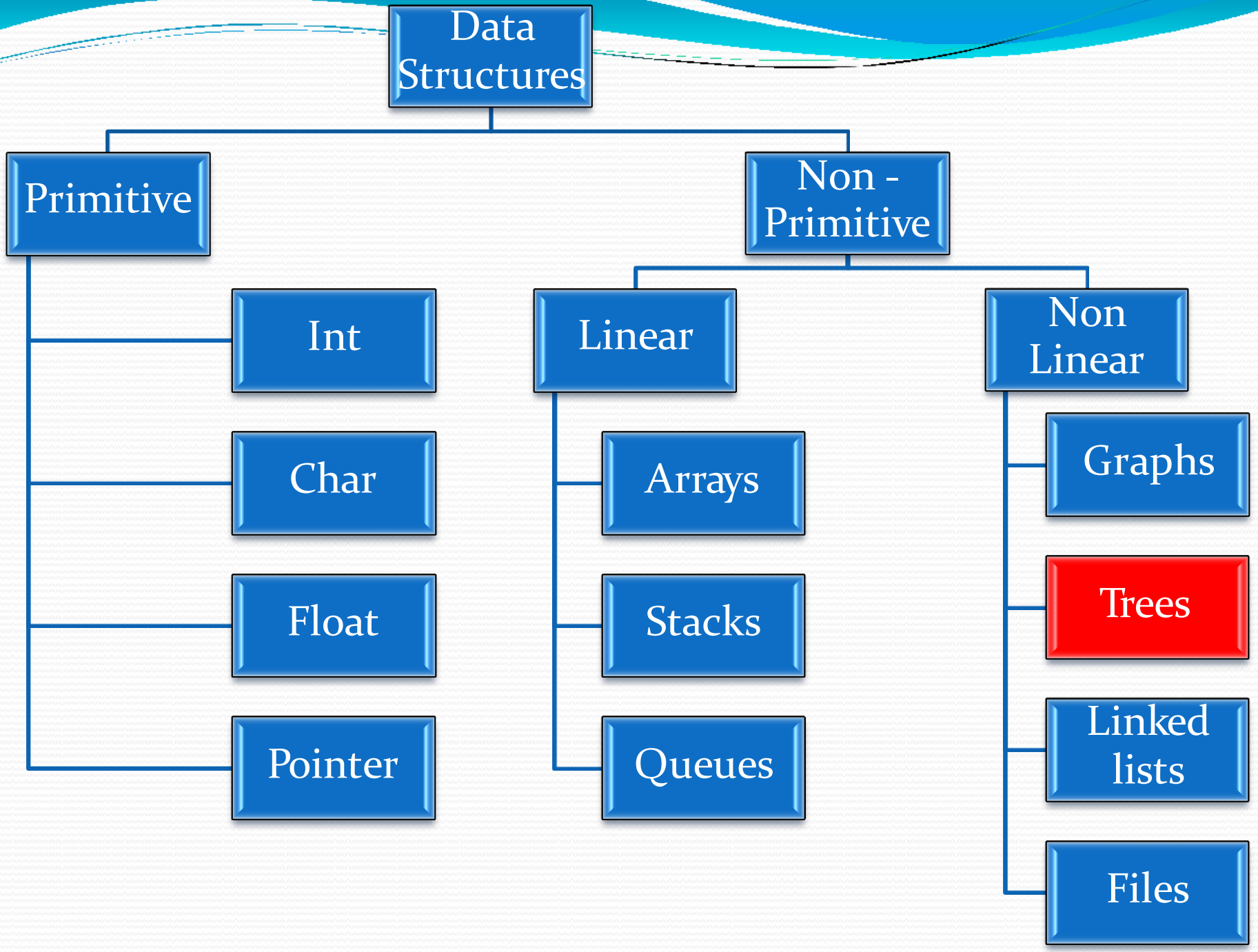
DATA STRUCTURES
UNIT 3 - TREES


Data Structures
Primitive
Non - Primitive
Int
Char
Float
Pointer
Linear
Non Linear
Arrays
Stacks
Queues
Graphs
Trees
Linked lists
Files

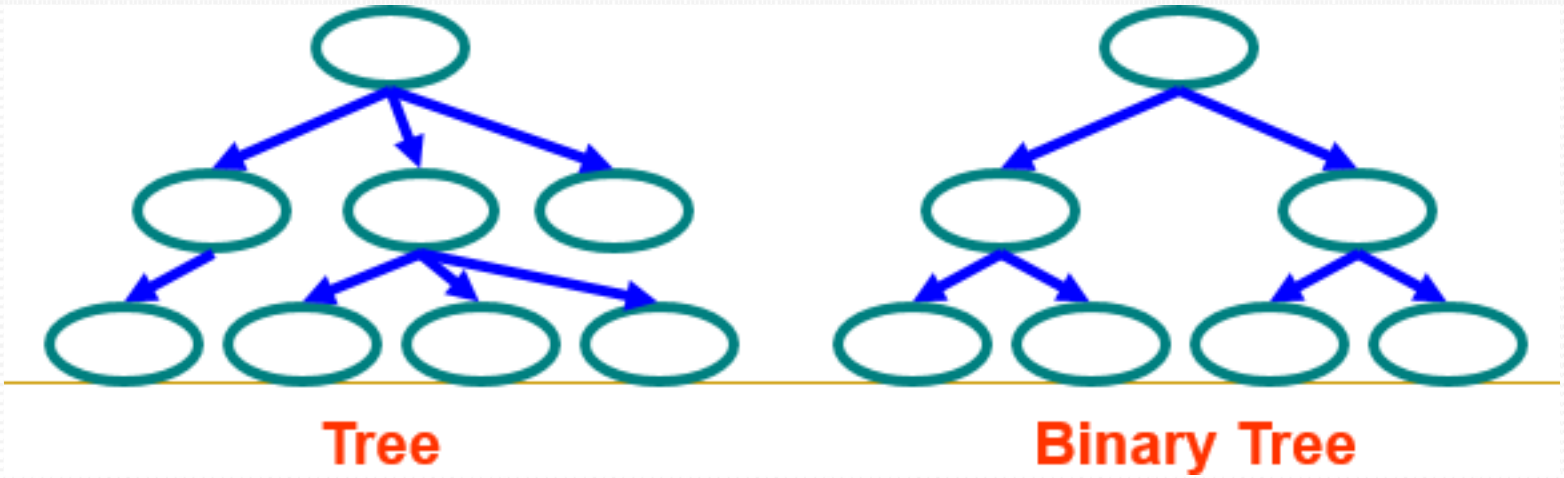

Data Structures
Primitive
Non - Primitive
Int
Char
Float
Pointer
Linear
Arrays
Stacks
Queues
Non Linear
Graphs
Trees
Linked lists
Files

Tree

- Each node can have 0 or more children
- A node can have at most one parent

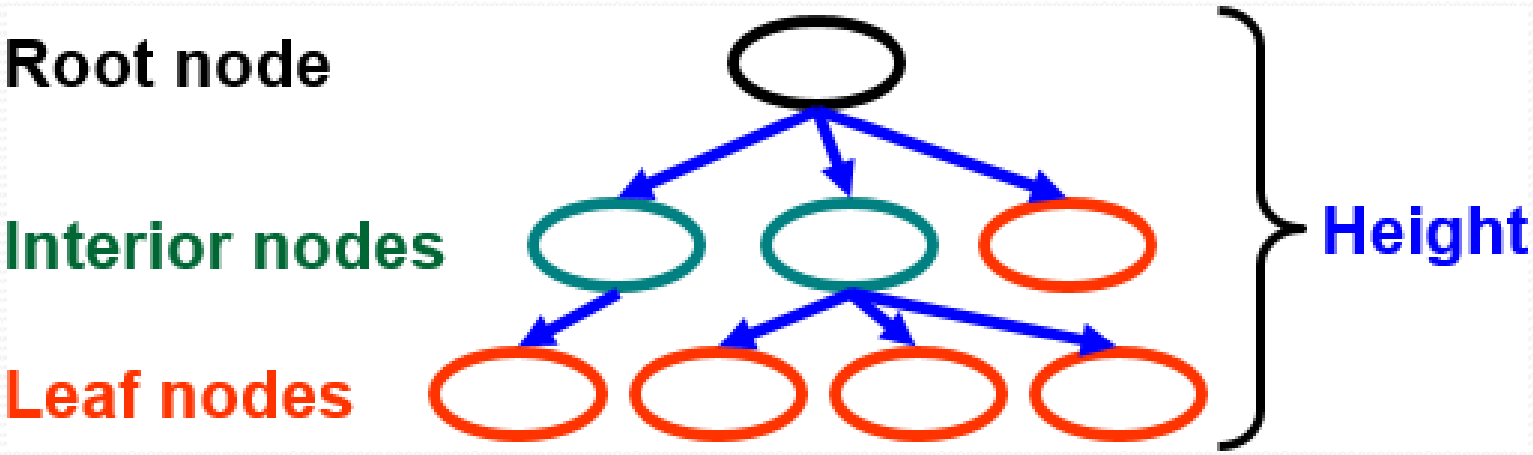
Binary tree

- Tree with 0–2 children per node
- Also known as Decision Making Tree



Terminology

- Root $\Rightarrow$ no parent
- Leaf $\Rightarrow$ no child
- Interior $\Rightarrow$ non-leaf
- Height $\Rightarrow$ distance from root to leaf (H)



Binary Tree Representation

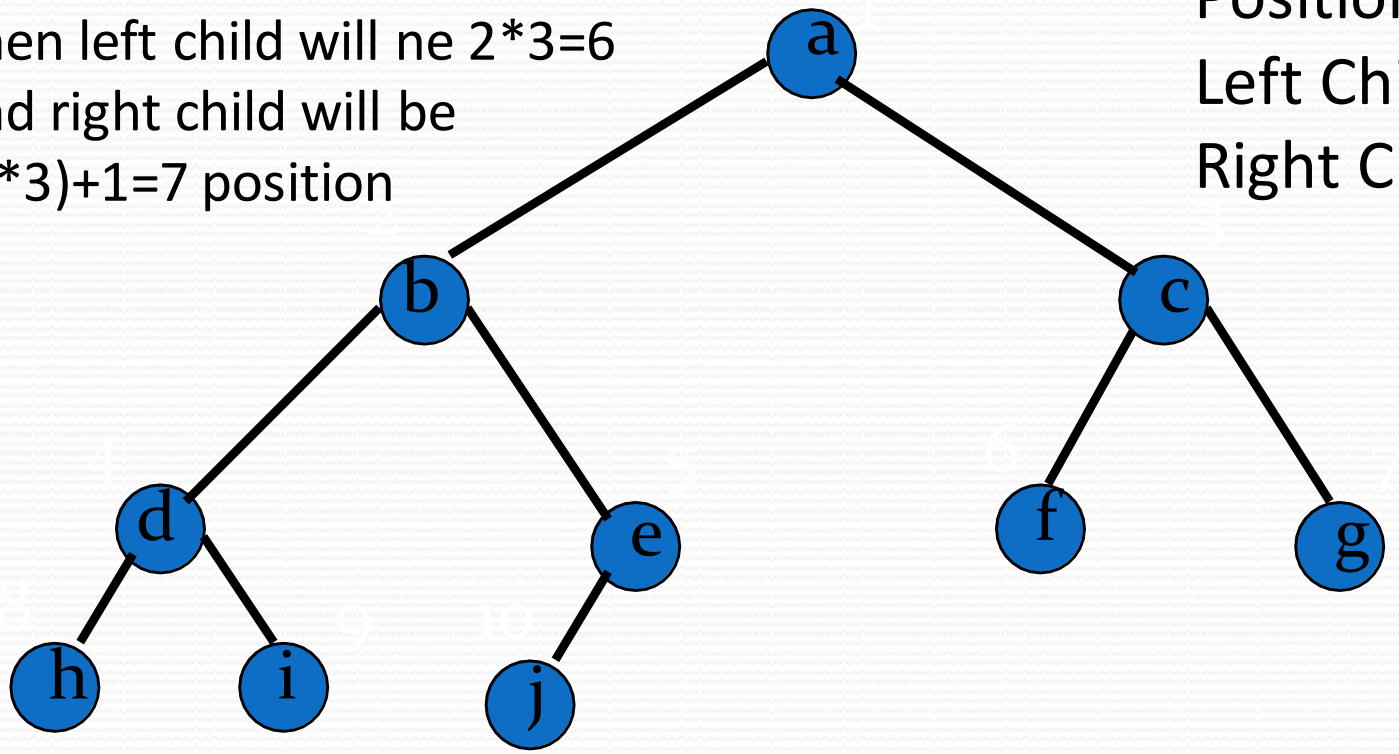
- Array representation.
- Linked representation.

Array Representation

- Number the nodes using the numbering scheme for a full binary tree. The node that is numbered i is stored in tree[i].

Example c is at position no. 3
Then left child will ne 2*3=6
and right child will be
(2*3)+1=7 position

If parent is at i^{th}
Position then
Left Child =2i
Right Child=2i+1



tree[]

	a	b	c	d	e	f	g	h	i	j	

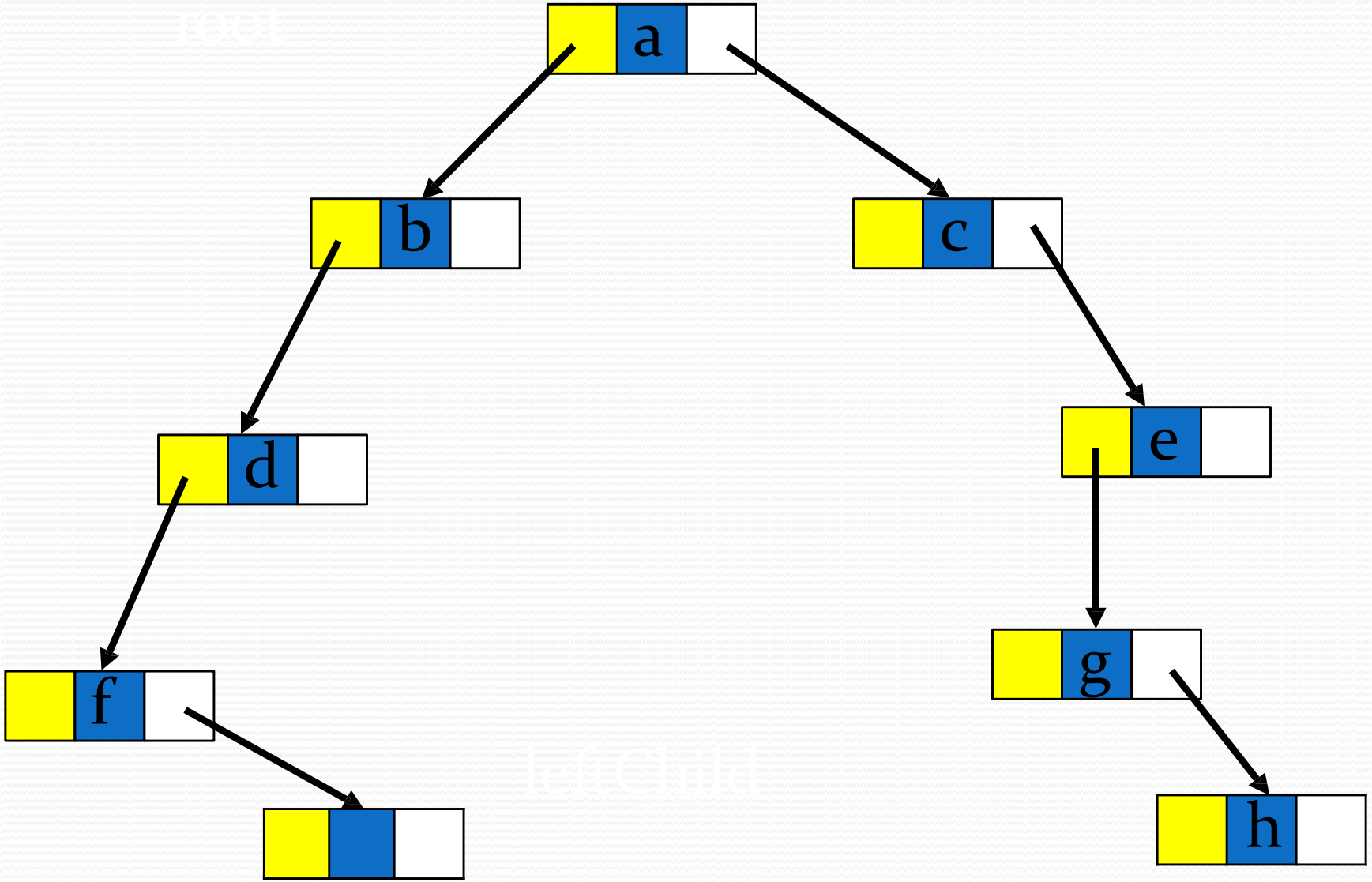
Linked Representation

- Each binary tree node is represented as an object whose data type is BinaryTreeNode.
- The space required by an n node binary tree is n * (space required by one node).

Link Representation of Binary Tree

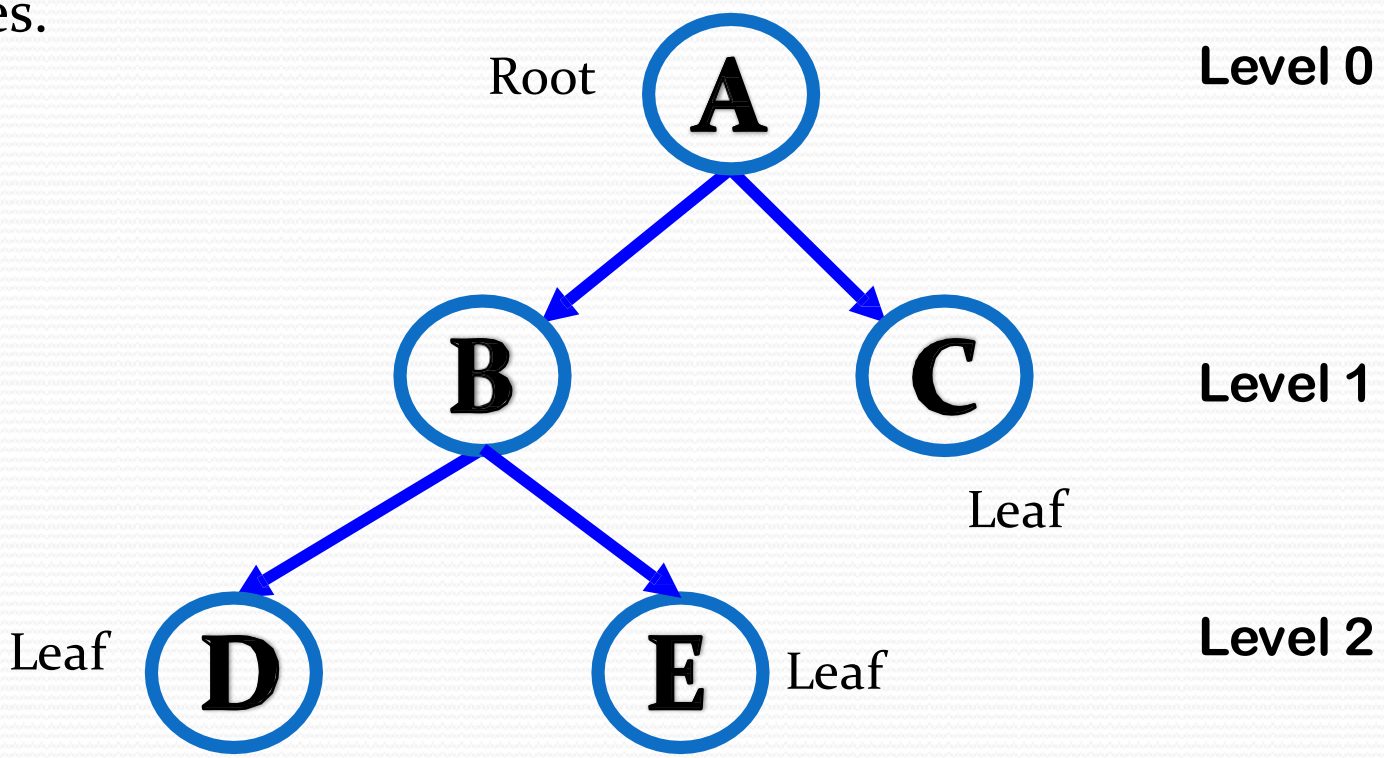
```
typedef struct node
{
    int data;
    struct node *lc,*rc;
};
```

Linked Representation Example

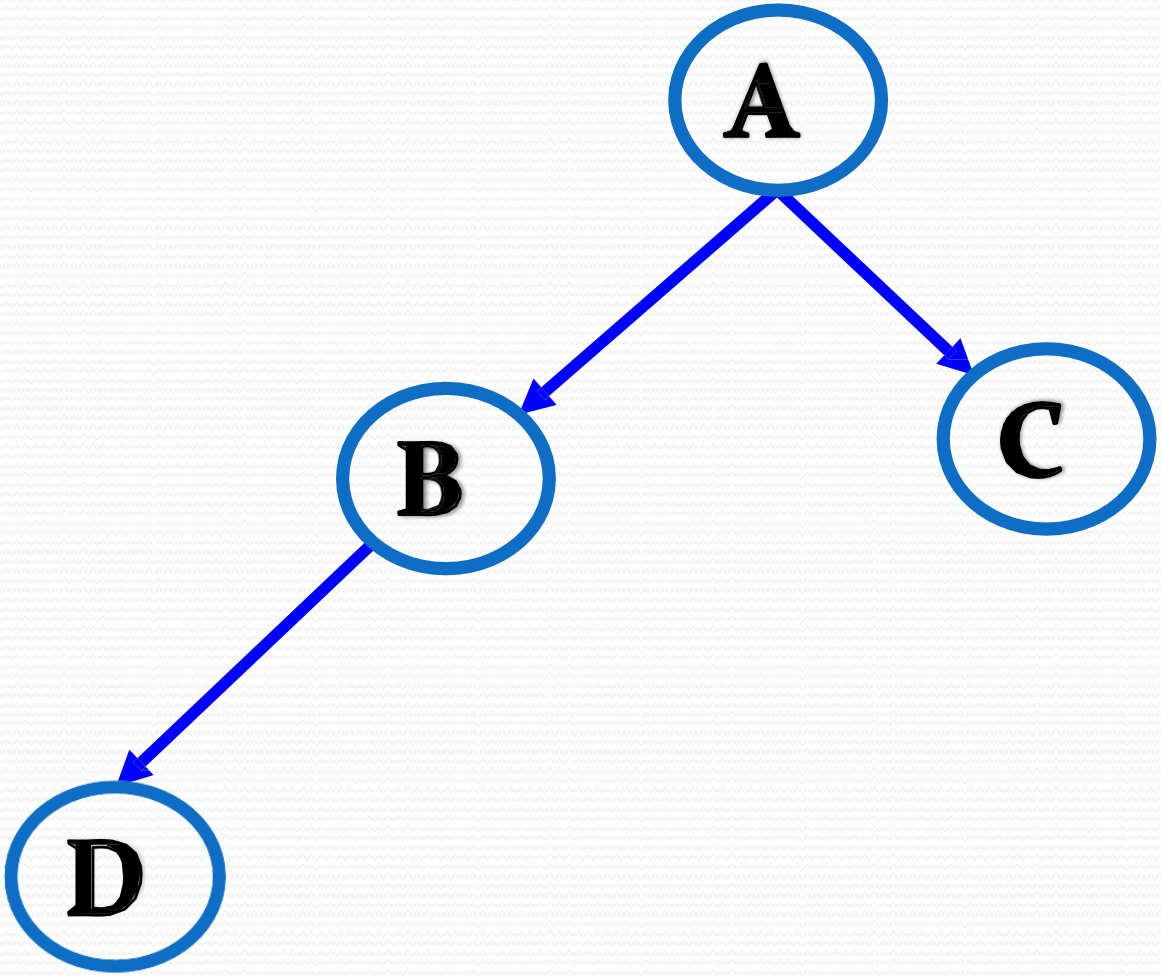


TREE

- A **tree** is a hierarchical representation of a finite set of one or more data items such that:
  - There is a special node called the root of the tree.
  - The nodes other than the root node form an ordered pair of disjoint subtrees.



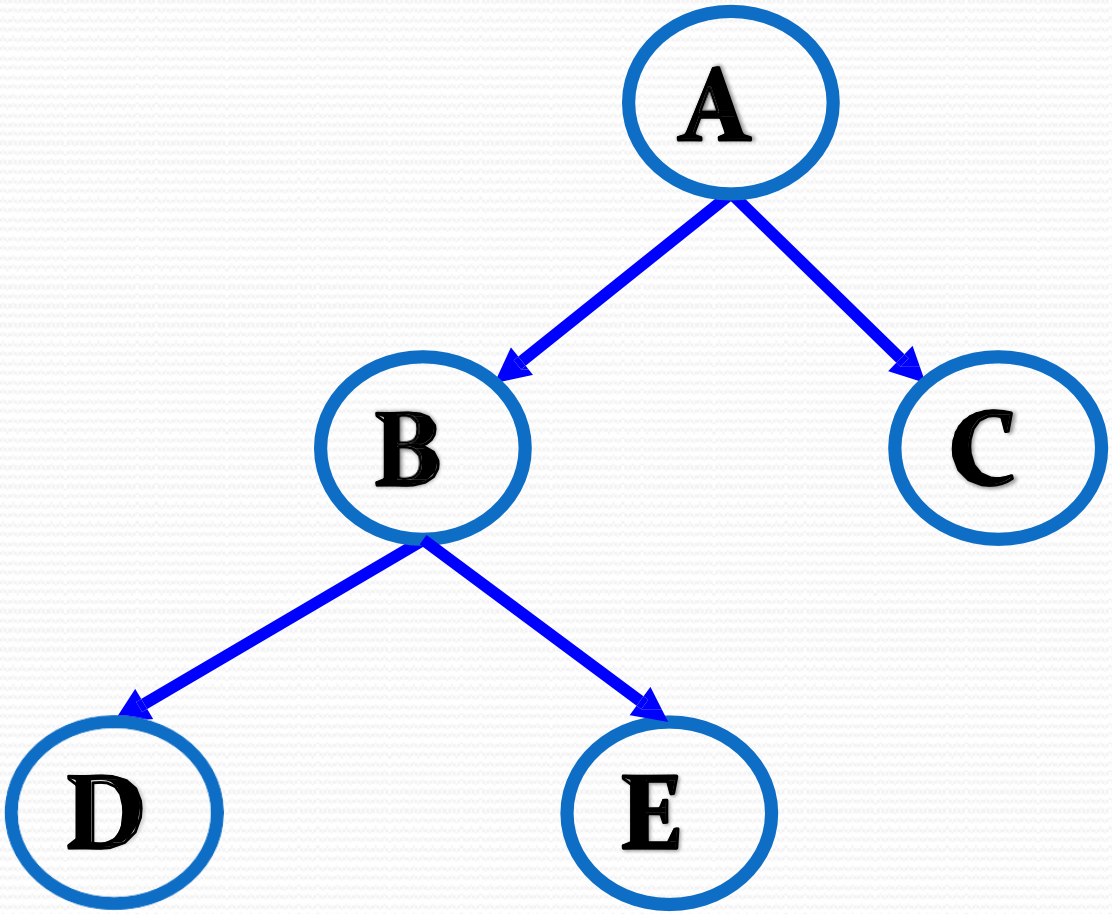
Binary Tree



Binary Tree

Binary Tree is a rooted tree in which root can have maximum two children such that each of them is again a binary tree. That means, there can be 0,1, or 2 children of any node.

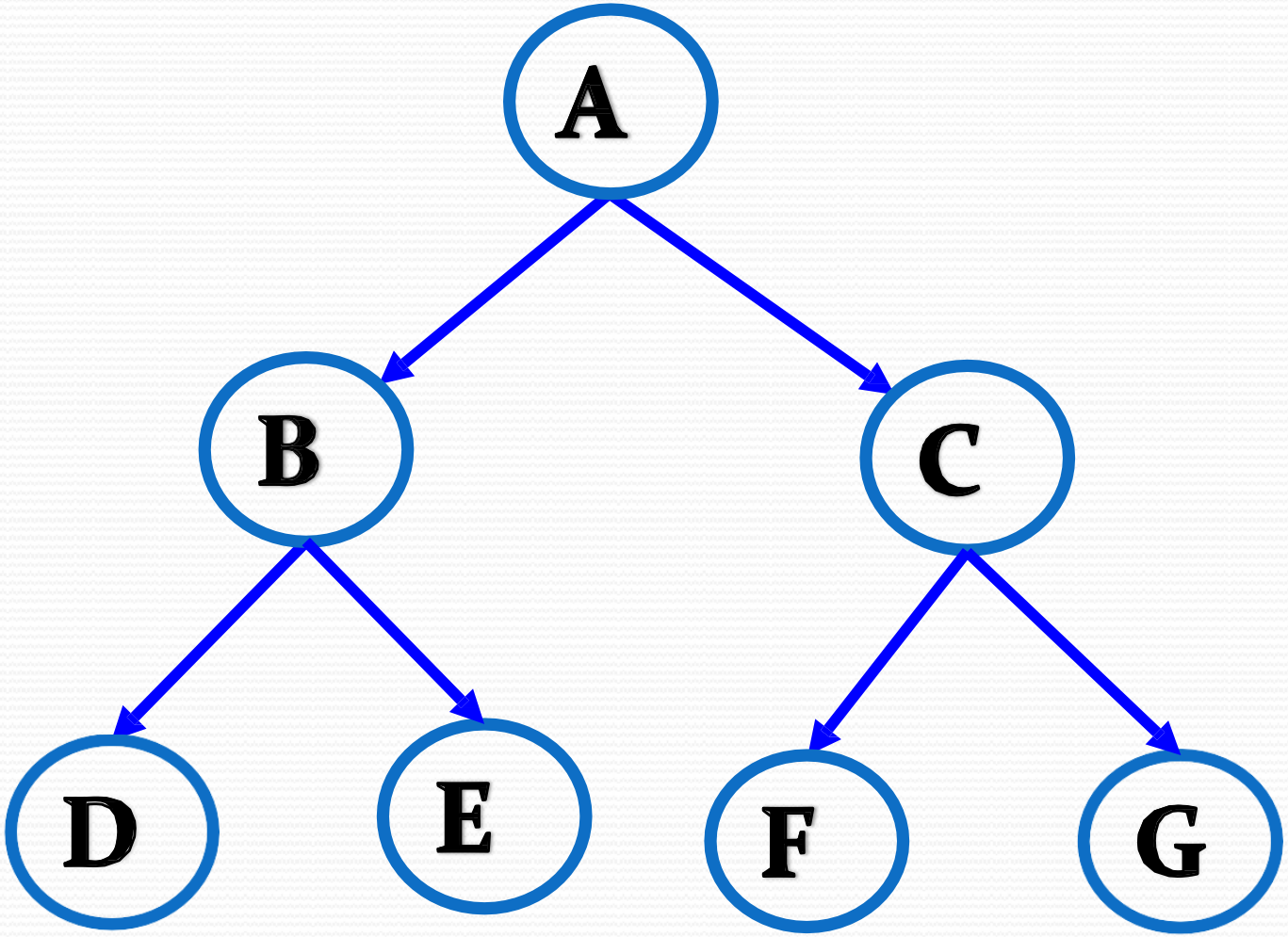
Strict Binary Tree



Strict Binary Tree

Strict Binary Tree is a Binary tree in which root can have exactly two children or no children at all. That means, there can be 0 or 2 children of any node.

Complete Binary Tree

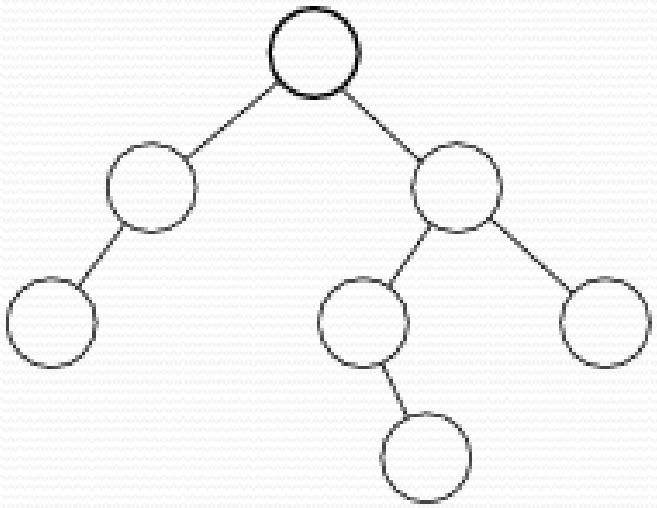


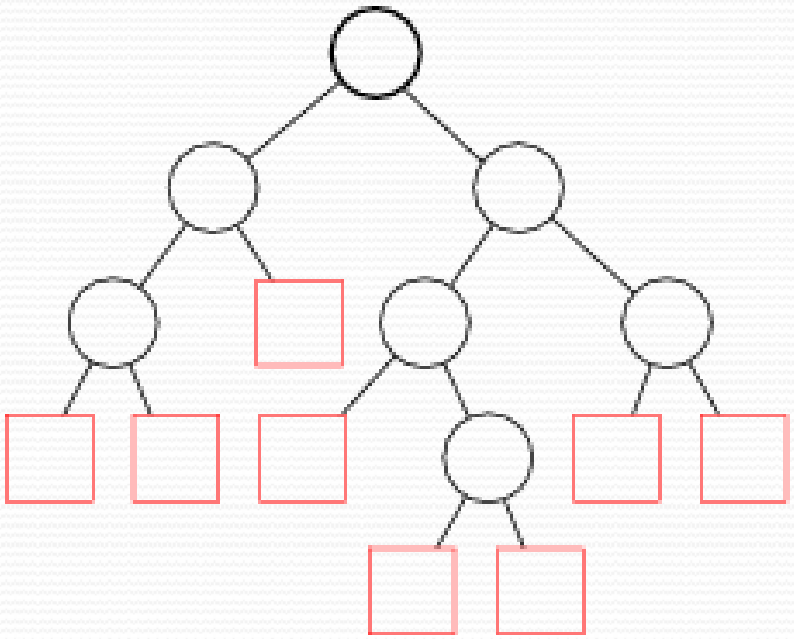
Complete Binary Tree

Complete Binary Tree is a Strict Binary tree in which every leaf node is at same level. That means, there are equal number of children in right and left subtree for every node.

Extended Binary Tree

- A *binary tree* with special *nodes* replacing every *null* subtree. Every regular node has two *children*, and every special node has nochildren.

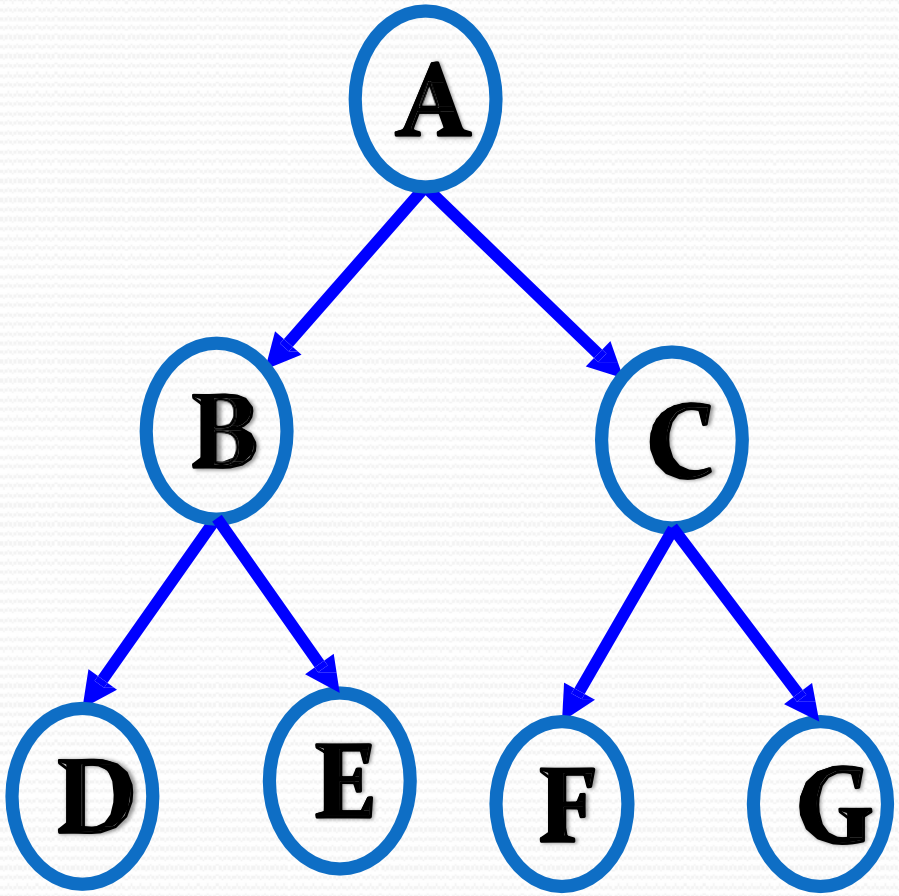




Extended Binary Tree

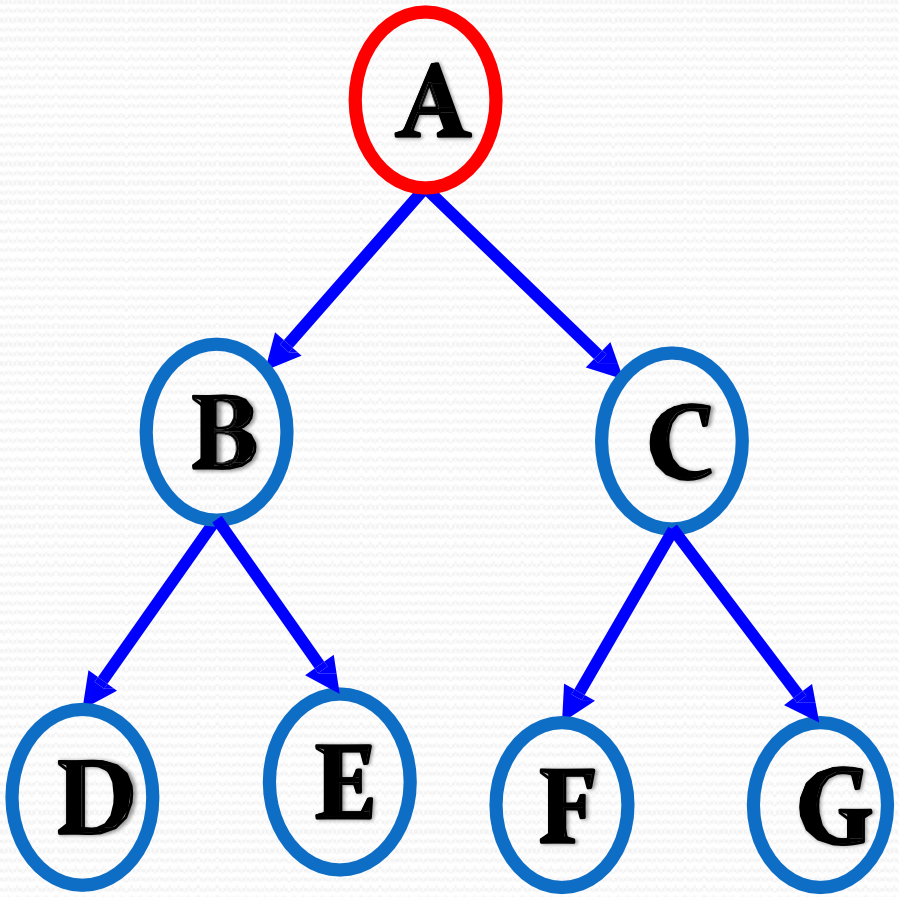
- An **extended binary tree** is a transformation of any **binary tree** into a complete **binary tree**.
- This transformation consists of replacing every null subtree of the original **tree** with "special nodes."
- The nodes from the original **tree** are then called as internal nodes, while the "special nodes" are called as external nodes.

Tree Traversal : Pre order



- Pre order (V L R)

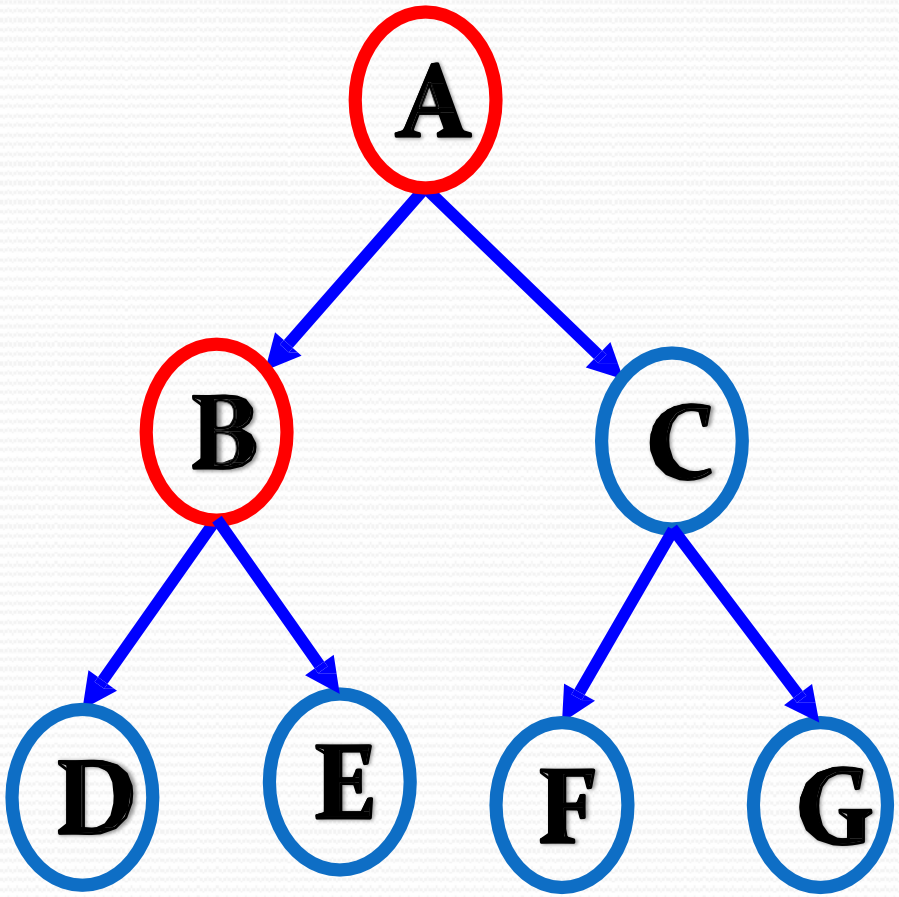
Tree Traversal : Pre order



- Pre order (V L R)

  A

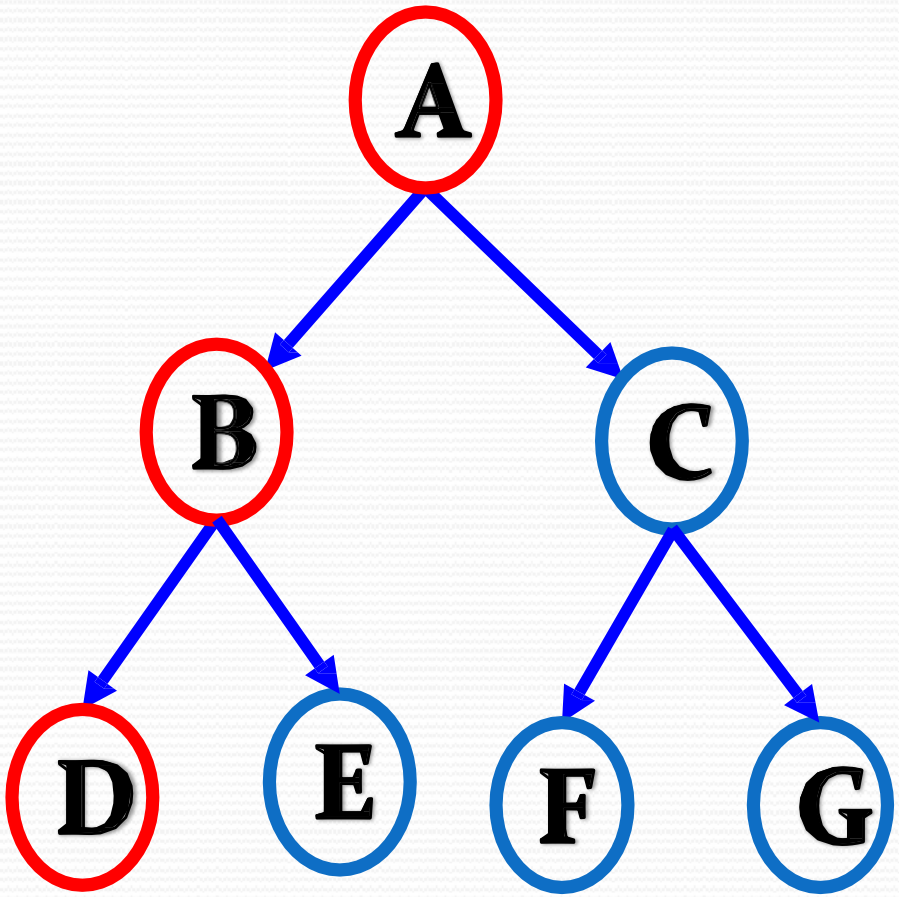
Tree Traversal : Pre order



- Pre order (V L R)

  A, B

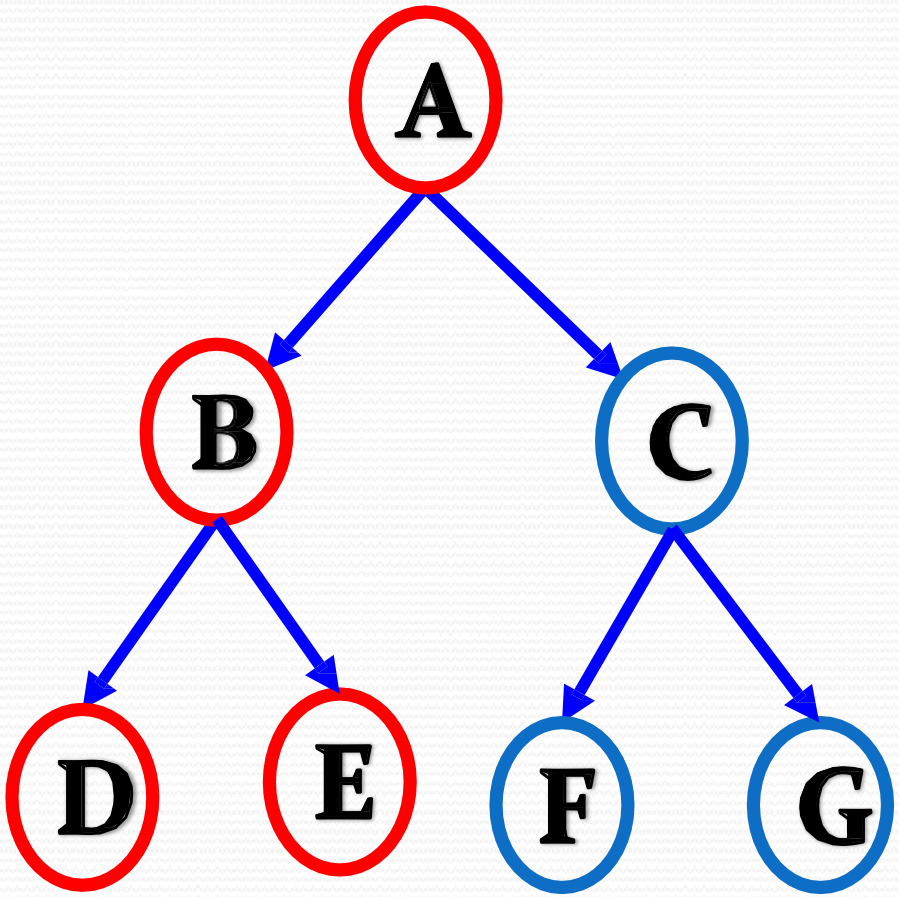
Tree Traversal : Pre order



- Pre order (V L R)

  A, B, D

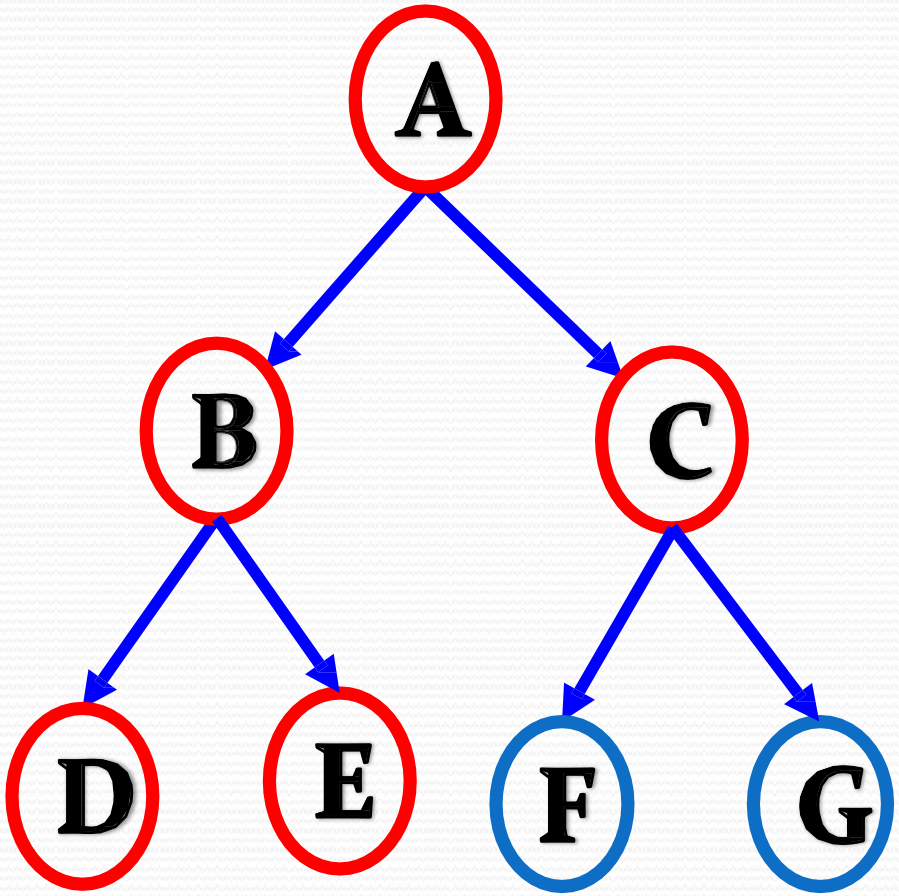
Tree Traversal : Pre order



- Pre order (V L R)

  A, B, D, E

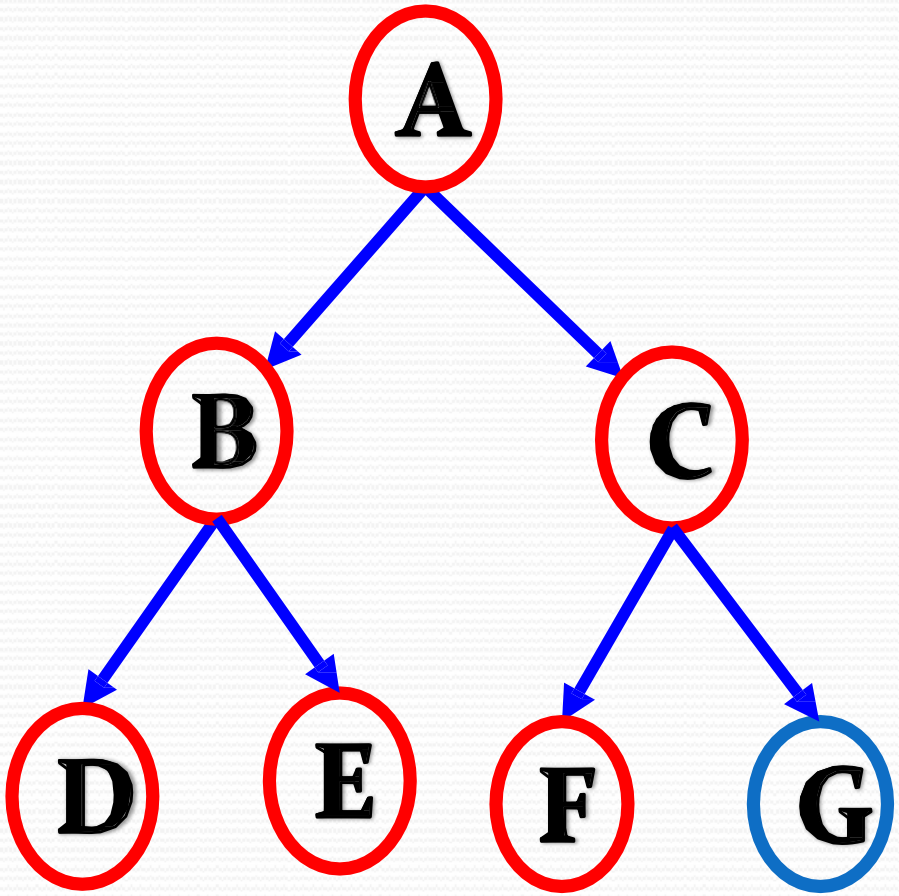
Tree Traversal : Pre order



- Pre order (V L R)

  A, B, D, E, C

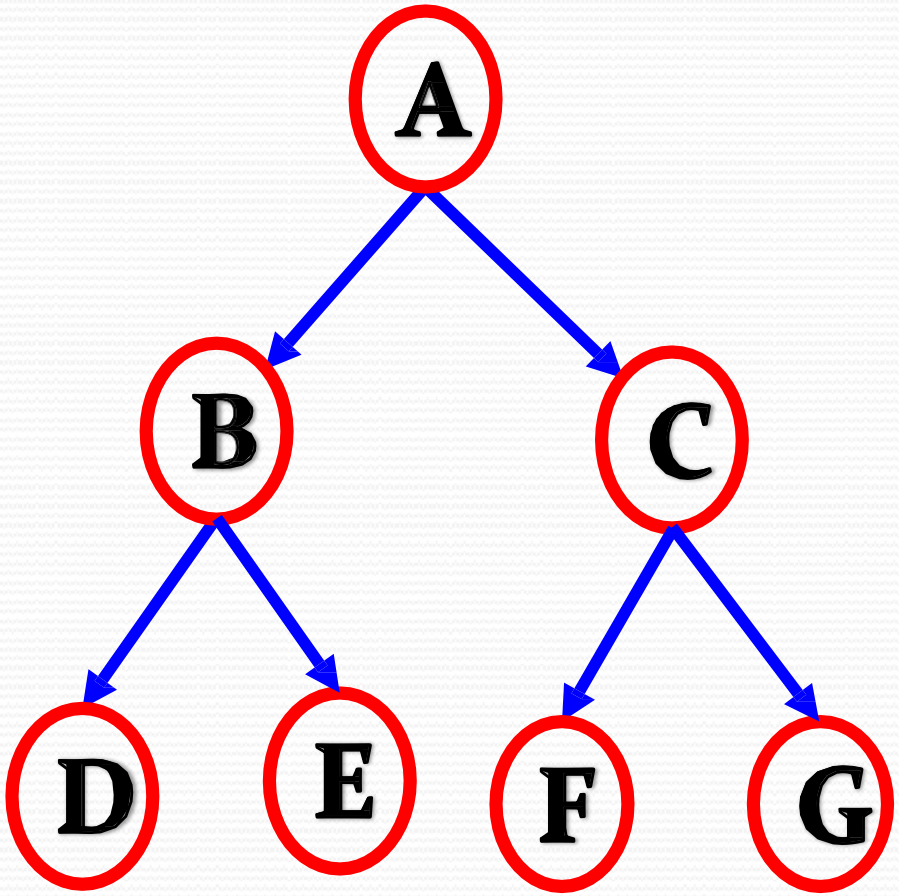
Tree Traversal : Pre order



- Pre order (V L R)

  A, B, D, E, C, F

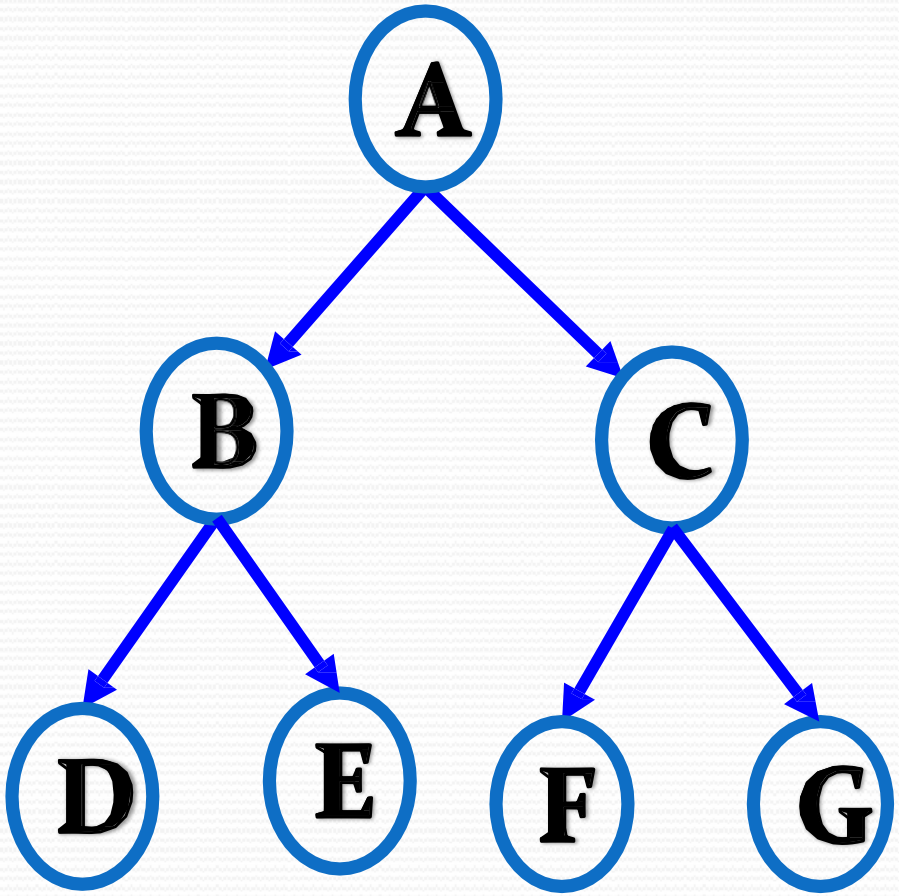
Tree Traversal : Pre order



- Pre order (V L R)

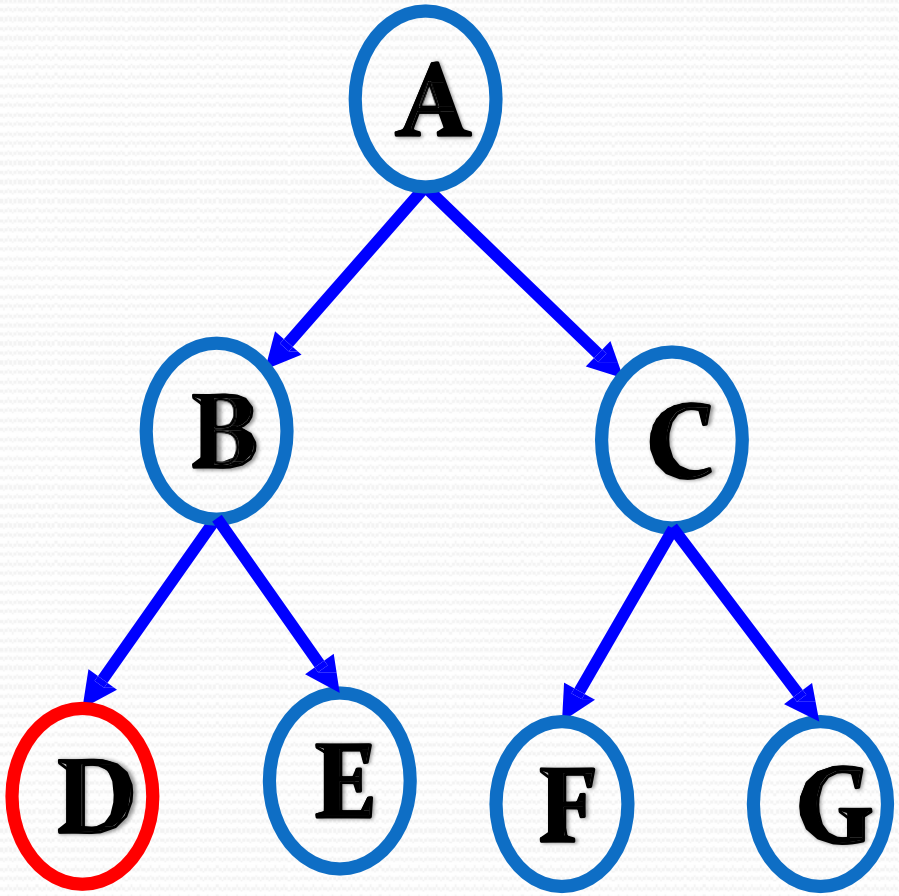
  A, B, D, E, C, F, G

Tree Traversal : In order



- In order (L V R)

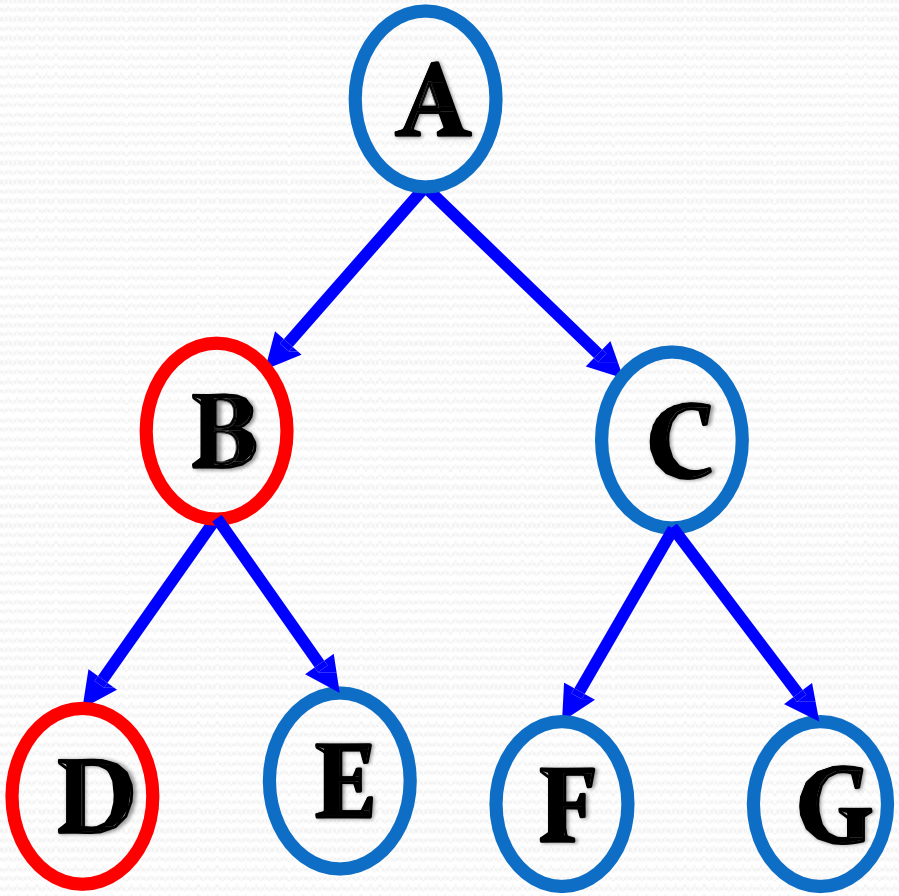
Tree Traversal : In order



- In order (L V R)

  D

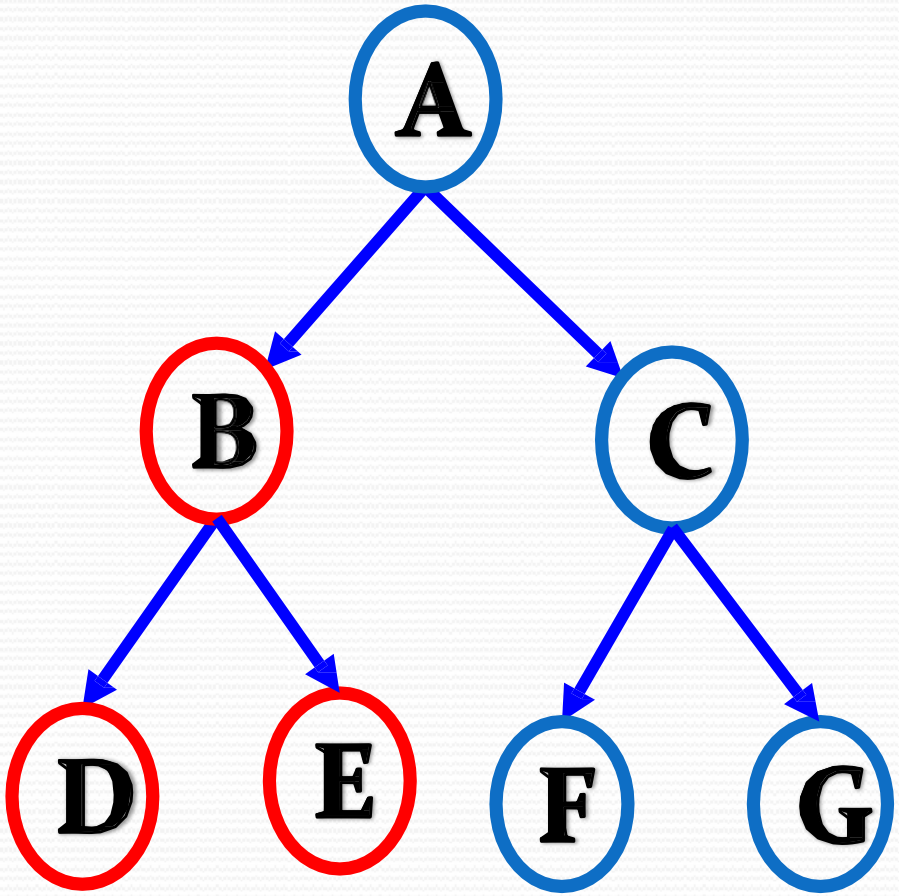
Tree Traversal : In order



- In order (L V R)

  D, B

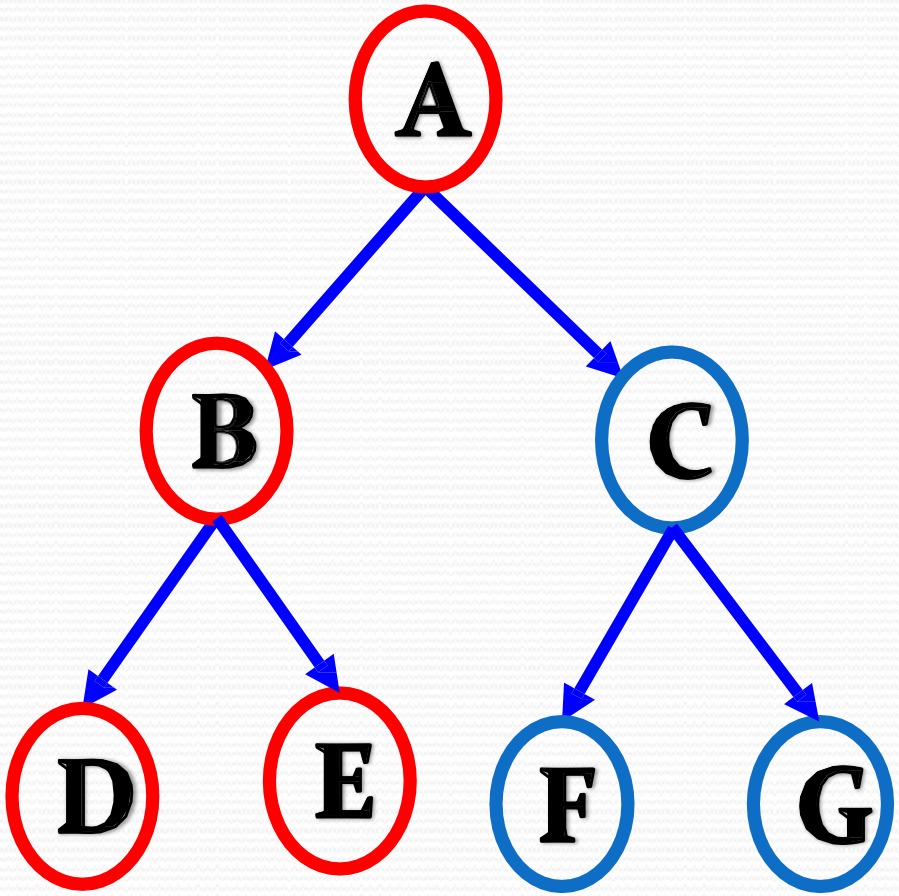
Tree Traversal : In order



- In order (L V R)

  D, B, E

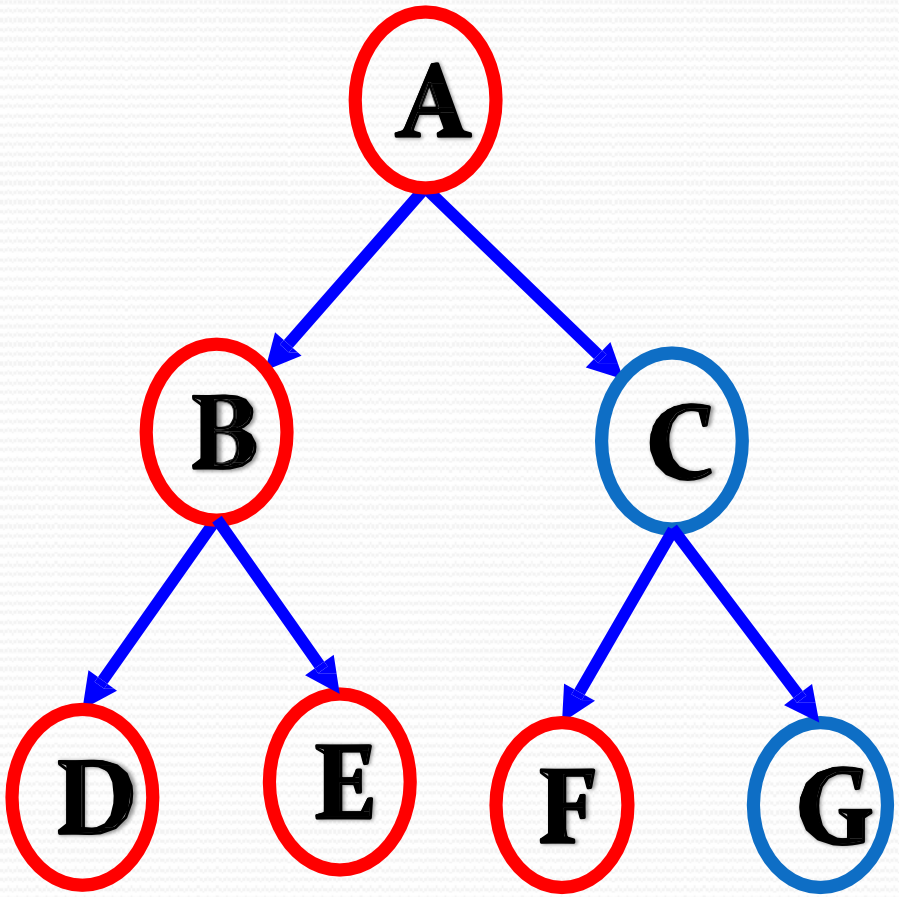
Tree Traversal : In order



- In order (L V R)

  D, B, E, A

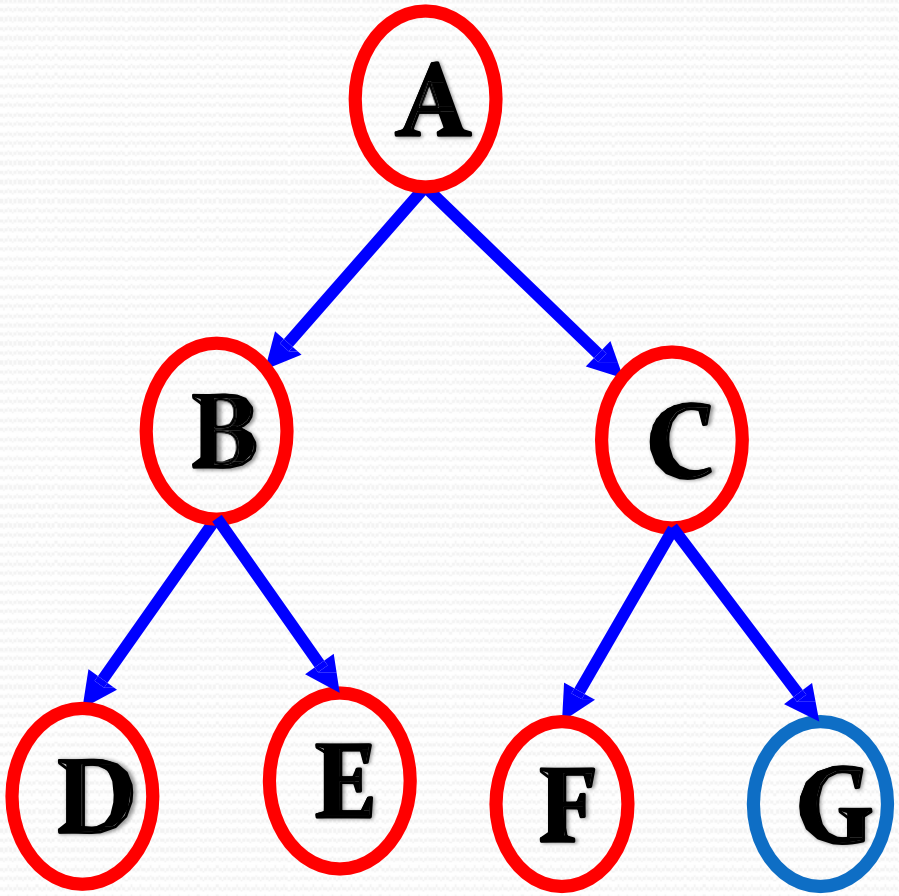
Tree Traversal : In order



- In order (L V R)

  D, B, E, A, F

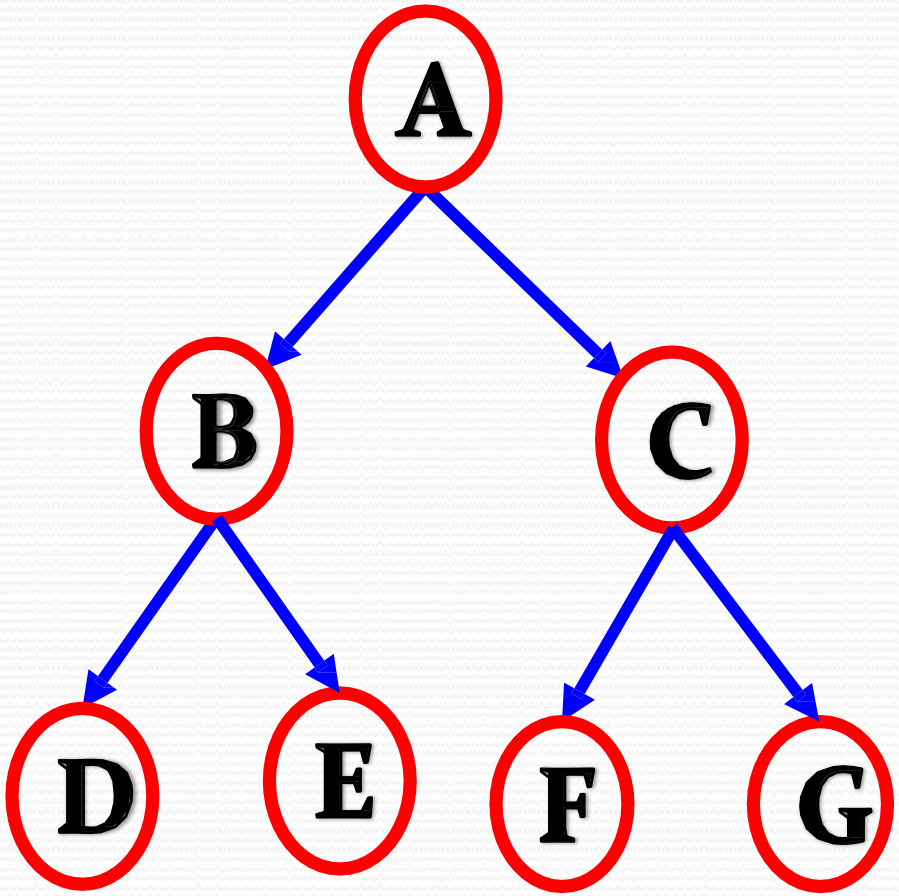
Tree Traversal : In order



- In order (L V R)

  D, B, E, A, F, C

Tree Traversal : In order

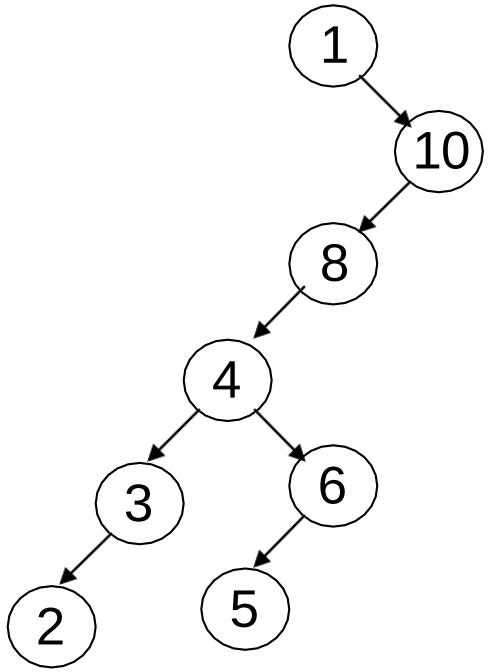


- In order (L V R)

  D, B, E, A, F, C, G

Inorder Traversal

1	10	8	4	6	3	2	5

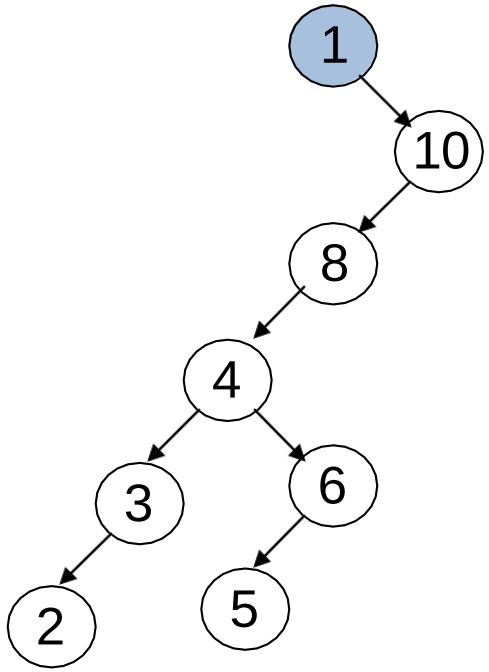


LVR

Left Visit Right

Inorder Traversal

1	10	8	4	6	3	2	5

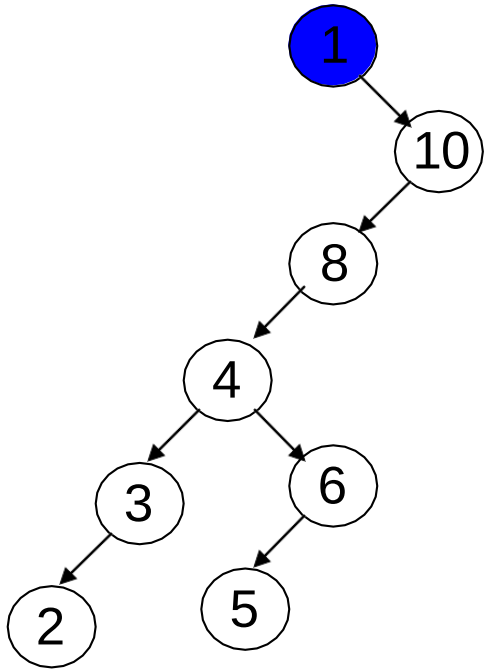


LVR

Left Visit Right

Inorder Traversal

1	10	8	4	6	3	2	5

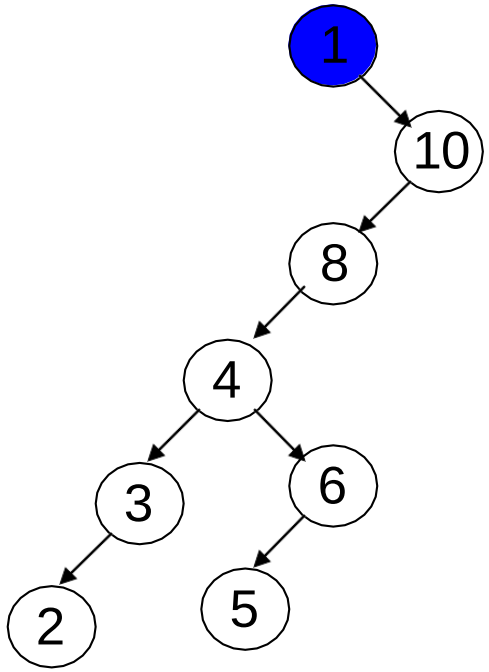


LVR

Left Visit Right

Inorder Traversal

1	10	8	4	6	3	2	5



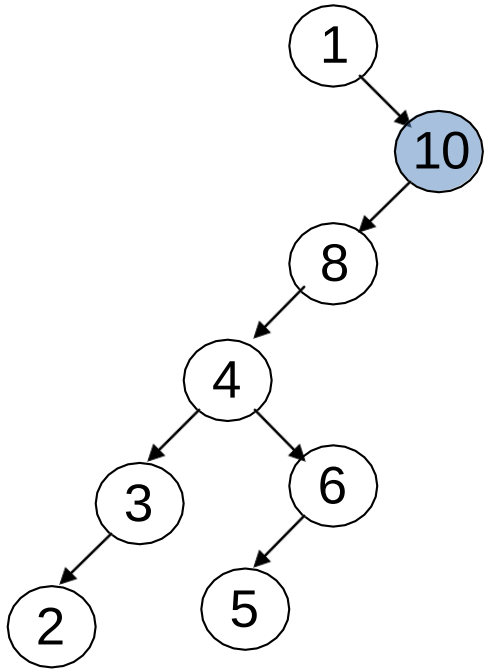
LVR

Left Visit Right

1

Inorder Traversal

1	10	8	4	6	3	2	5



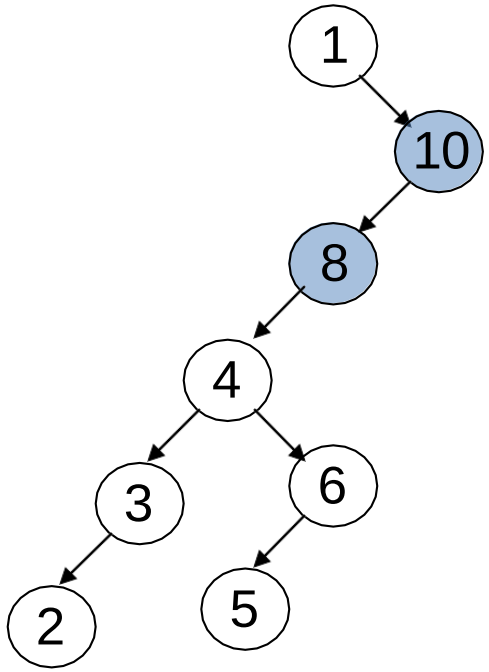
LVR

Left Visit Right

1

Inorder Traversal

1	10	8	4	6	3	2	5



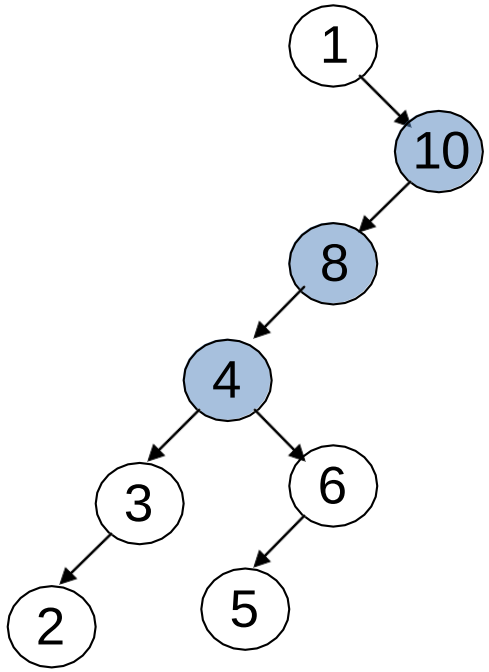
LVR

Left Visit Right

1

Inorder Traversal

1	10	8	4	6	3	2	5



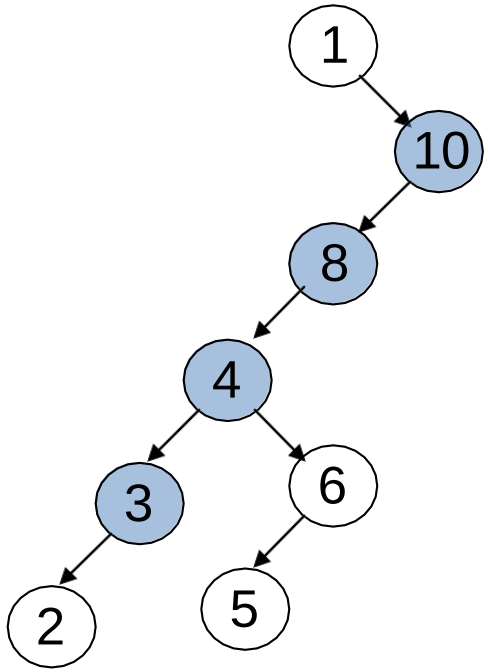
LVR

Left Visit Right

1

Inorder Traversal

1	10	8	4	6	3	2	5



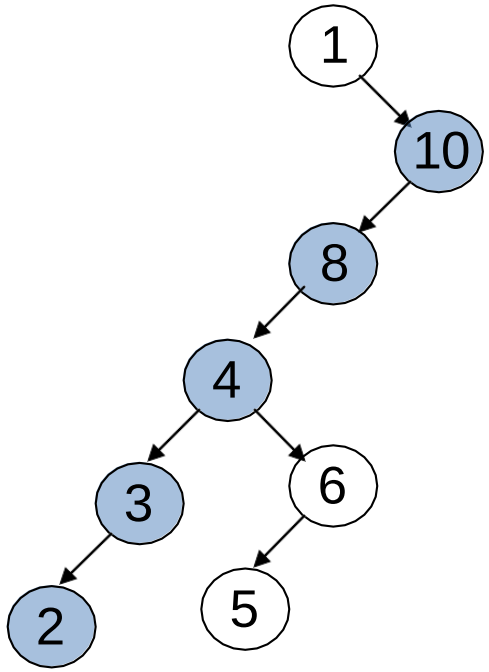
LVR

Left Visit Right

1

Inorder Traversal

1	10	8	4	6	3	2	5



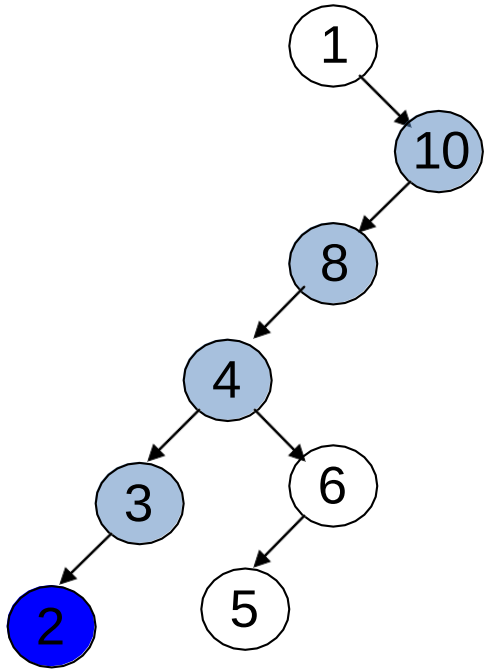
LVR

Left Visit Right

1

Inorder Traversal

1	10	8	4	6	3	2	5



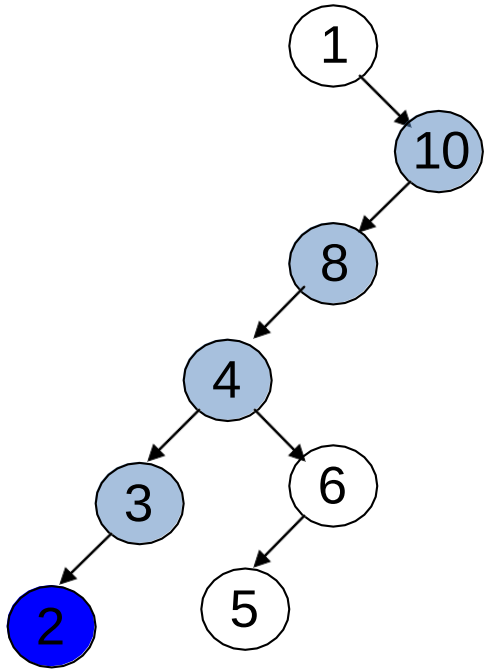
LVR

Left Visit Right

1

Inorder Traversal

1	10	8	4	6	3	2	5



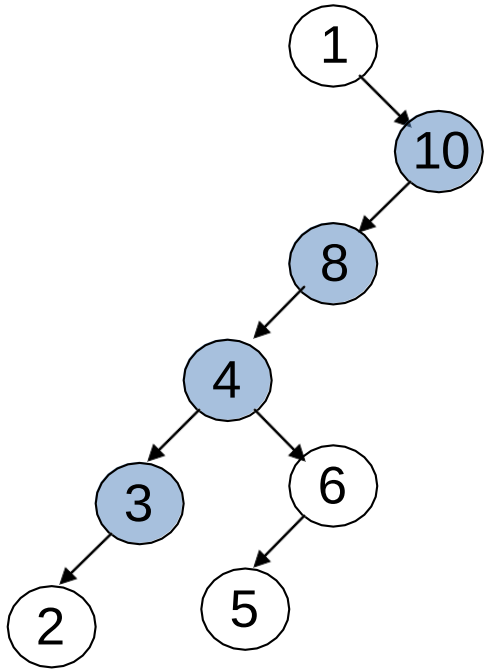
LVR

Left Visit Right

1	2

Inorder Traversal

1	10	8	4	6	3	2	5



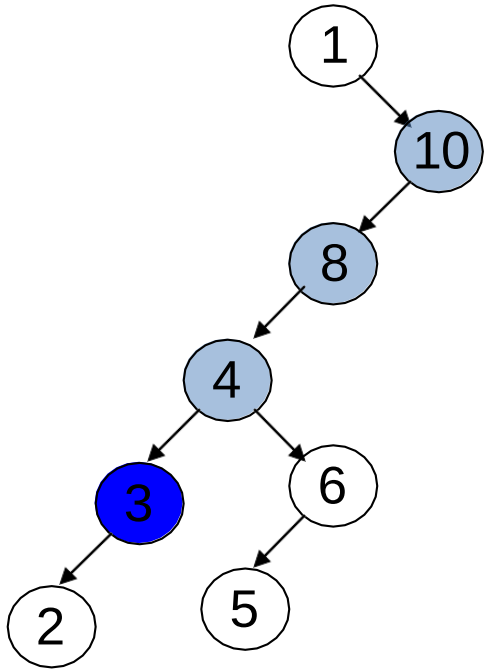
LVR

Left Visit Right

1	2

Inorder Traversal

1	10	8	4	6	3	2	5



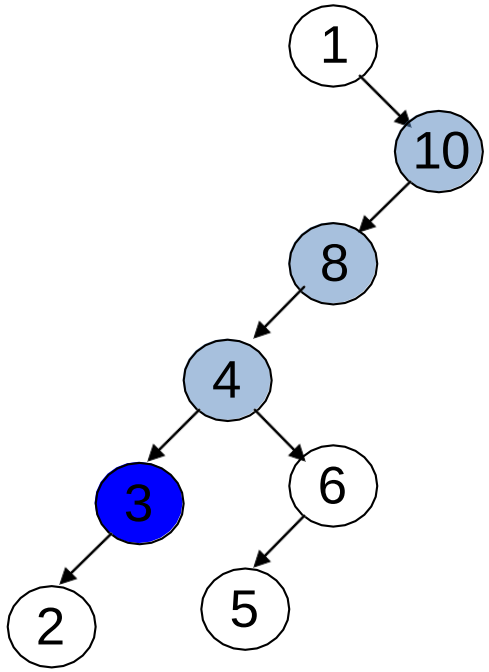
LVR

Left Visit Right

1	2

Inorder Traversal

1	10	8	4	6	3	2	5



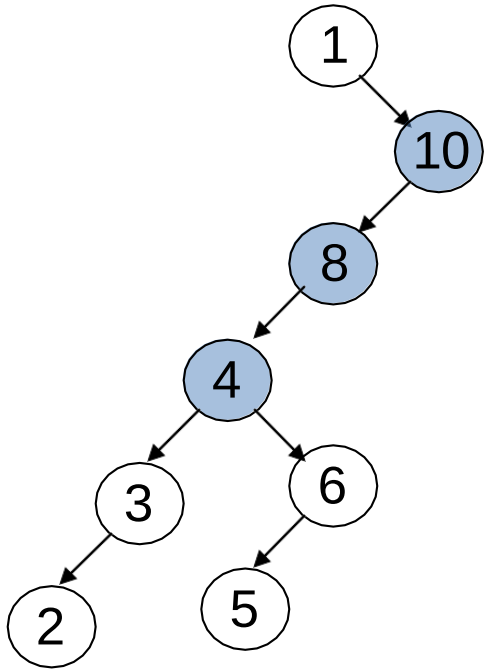
LVR

Left Visit Right

1	2	3

Inorder Traversal

1	10	8	4	6	3	2	5



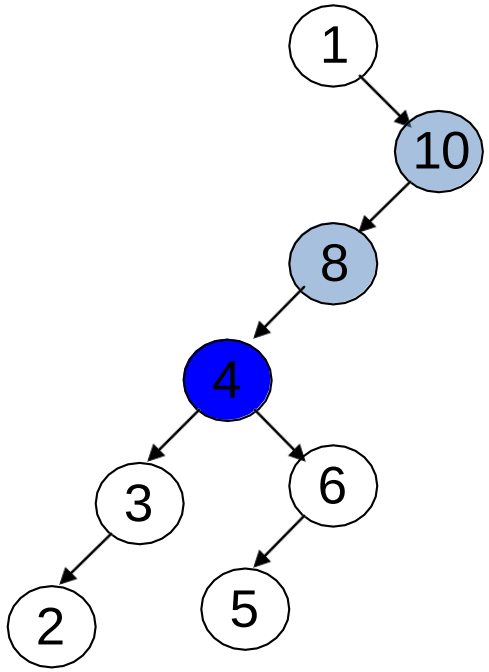
LVR

Left Visit Right

1	2	3

Inorder Traversal

1	10	8	4	6	3	2	5



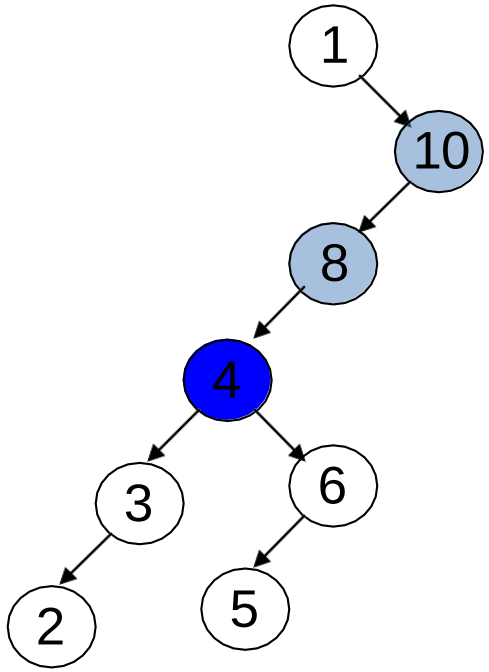
LVR

Left Visit Right

1	2	3

Inorder Traversal

1	10	8	4	6	3	2	5



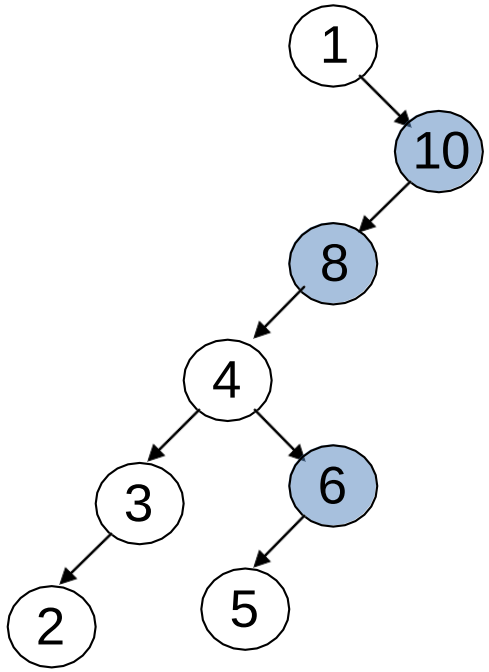
LVR

Left Visit Right

1	2	3	4

Inorder Traversal

1	10	8	4	6	3	2	5



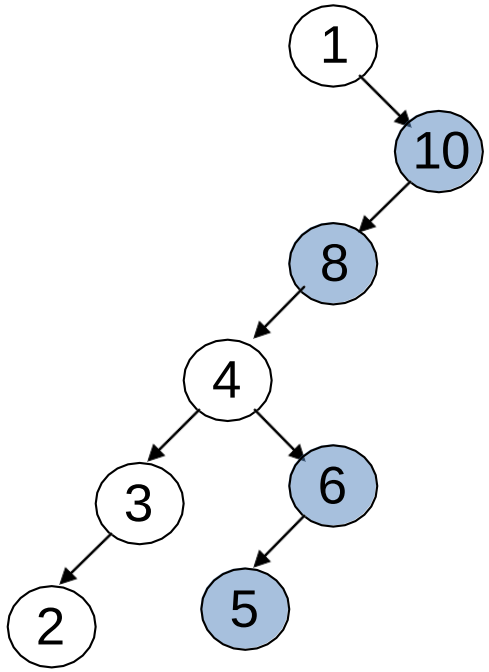
LVR

Left Visit Right

1	2	3	4

Inorder Traversal

1	10	8	4	6	3	2	5



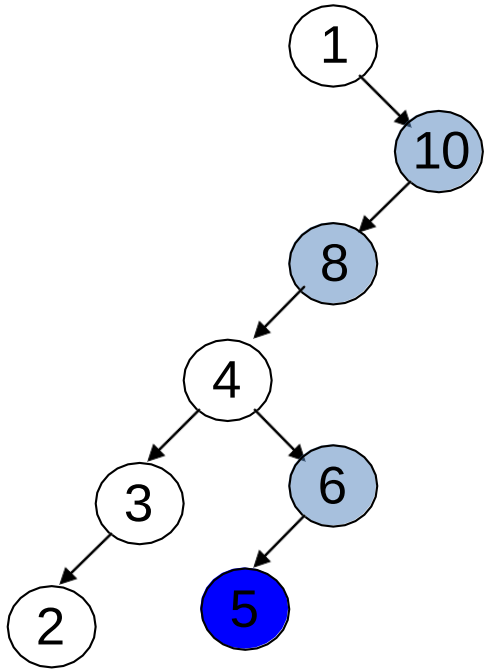
LVR

Left Visit Right

1	2	3	4

Inorder Traversal

1	10	8	4	6	3	2	5



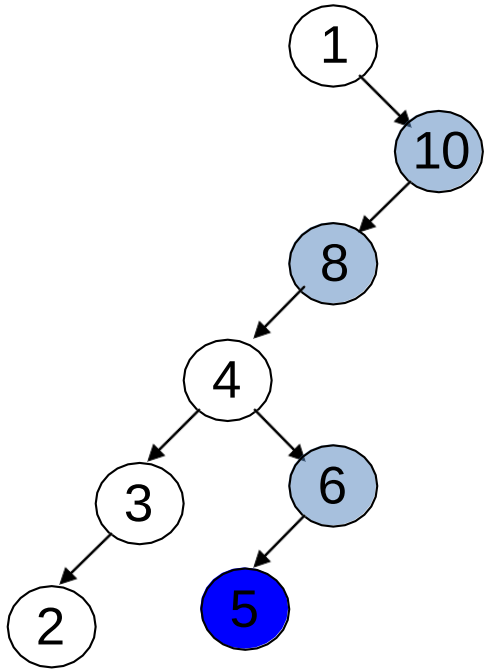
LVR

Left Visit Right

1	2	3	4

Inorder Traversal

1	10	8	4	6	3	2	5



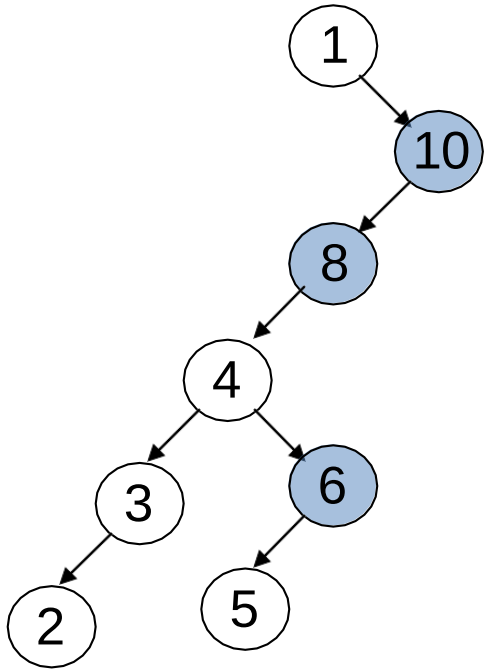
LVR

Left Visit Right

1	2	3	4	5

Inorder Traversal

1	10	8	4	6	3	2	5



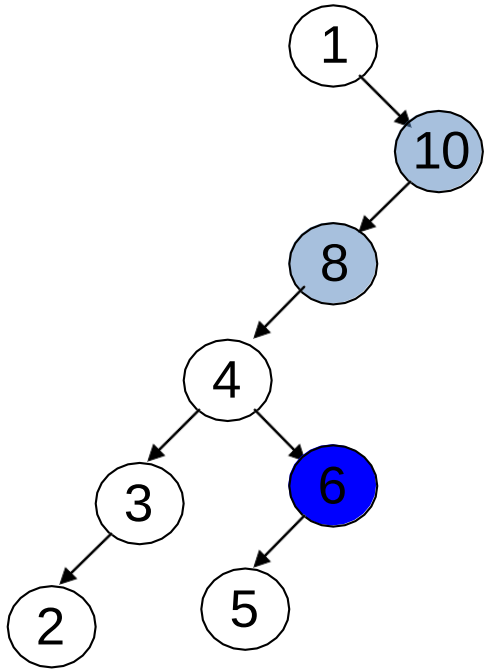
LVR

Left Visit Right

1	2	3	4	5

Inorder Traversal

1	10	8	4	6	3	2	5



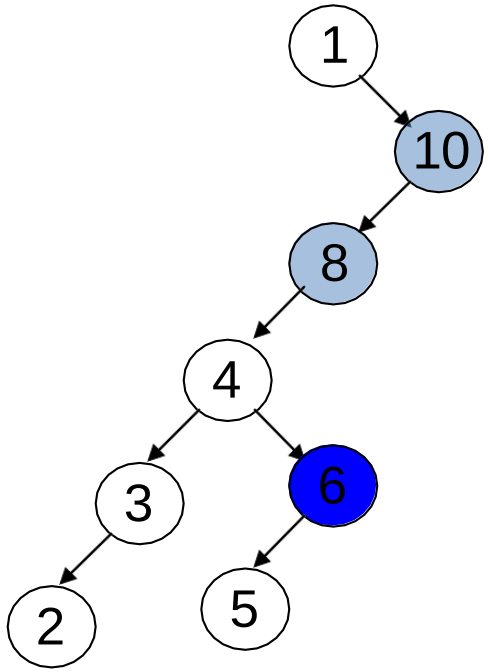
LVR

Left Visit Right

1	2	3	4	5

Inorder Traversal

1	10	8	4	6	3	2	5



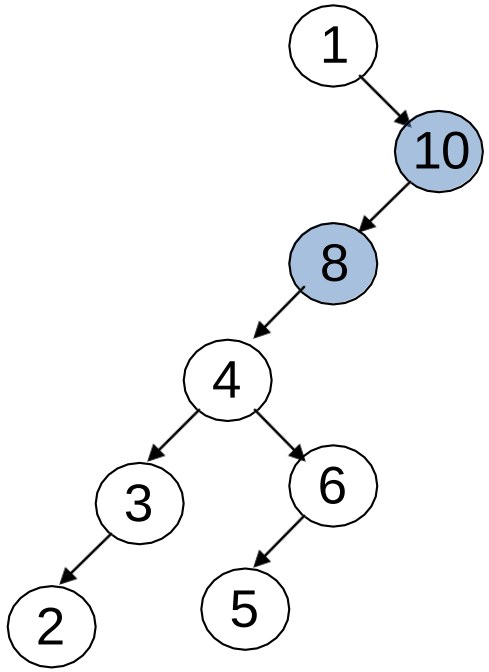
LVR

Left Visit Right

1	2	3	4	5	6

Inorder Traversal

1	10	8	4	6	3	2	5



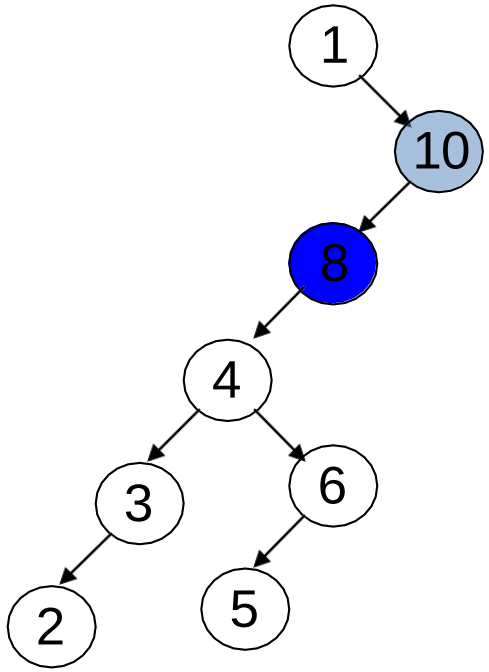
LVR

Left Visit Right

1	2	3	4	5	6

Inorder Traversal

1	10	8	4	6	3	2	5



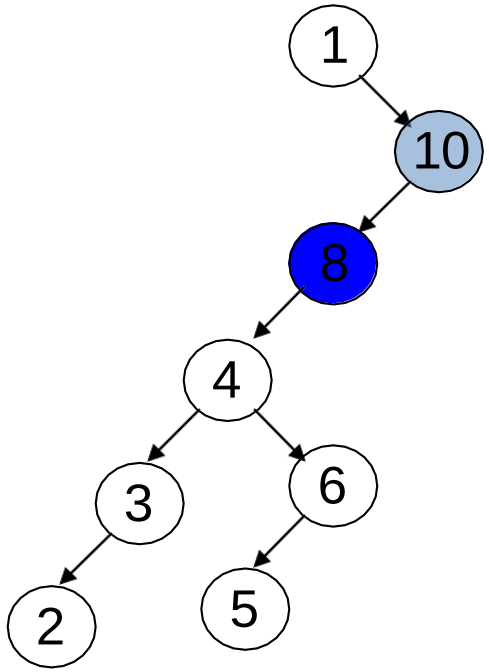
LVR

Left Visit Right

1	2	3	4	5	6

Inorder Traversal

1	10	8	4	6	3	2	5



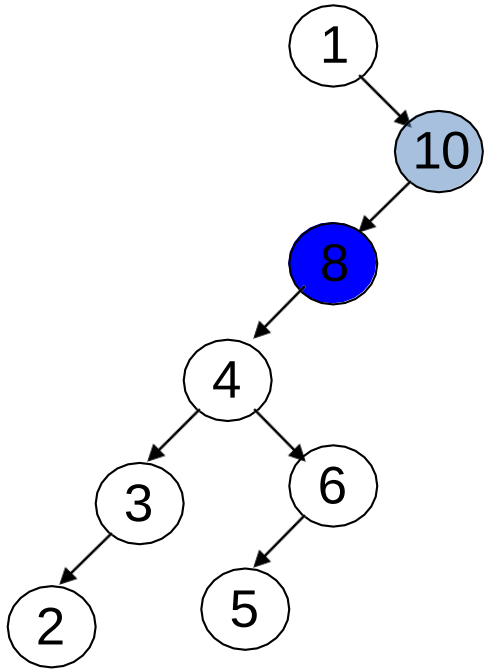
LVR

Left Visit Right

1	2	3	4	5	6

Inorder Traversal

1	10	8	4	6	3	2	5



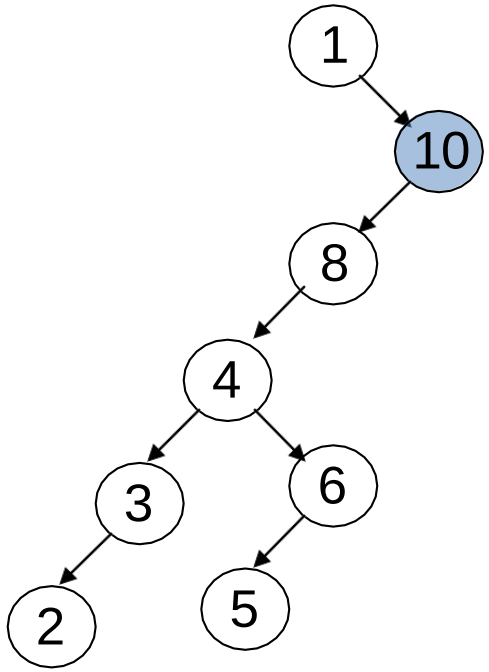
LVR

Left Visit Right

1	2	3	4	5	6	8

Inorder Traversal

1	10	8	4	6	3	2	5



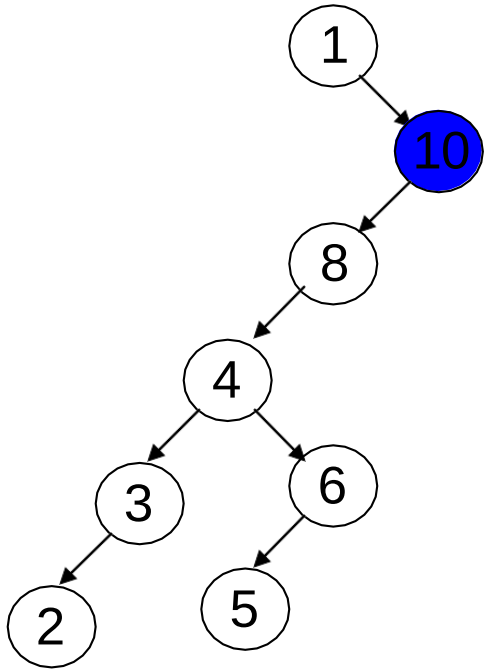
LVR

Left Visit Right

1	2	3	4	5	6	8

Inorder Traversal

1	10	8	4	6	3	2	5



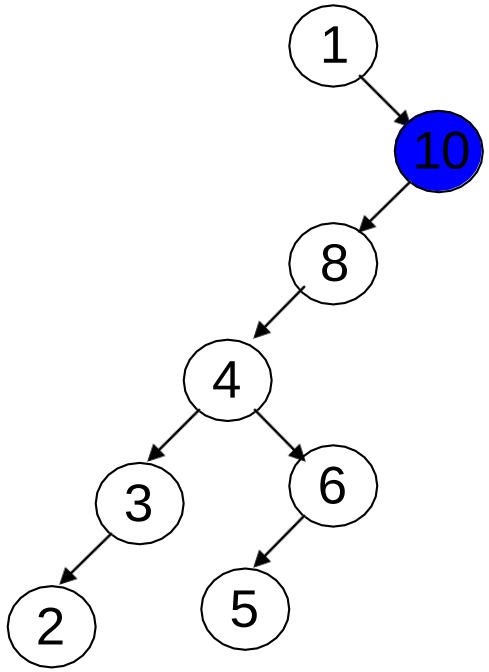
LVR

Left Visit Right

1	2	3	4	5	6	8

Inorder Traversal

1	10	8	4	6	3	2	5



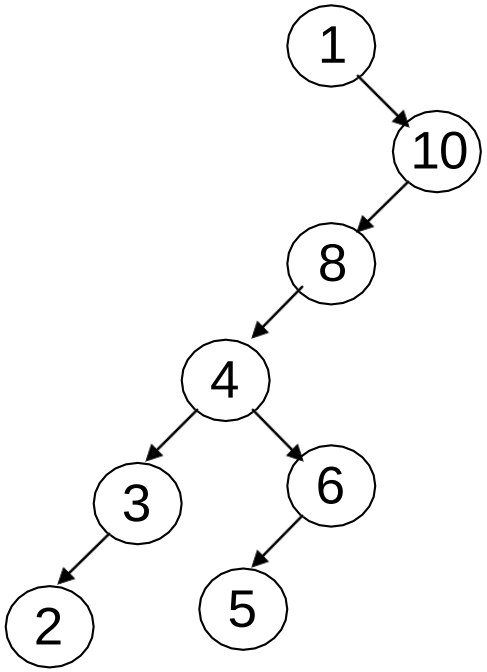
LVR

Left Visit Right

1	2	3	4	5	6	8	10

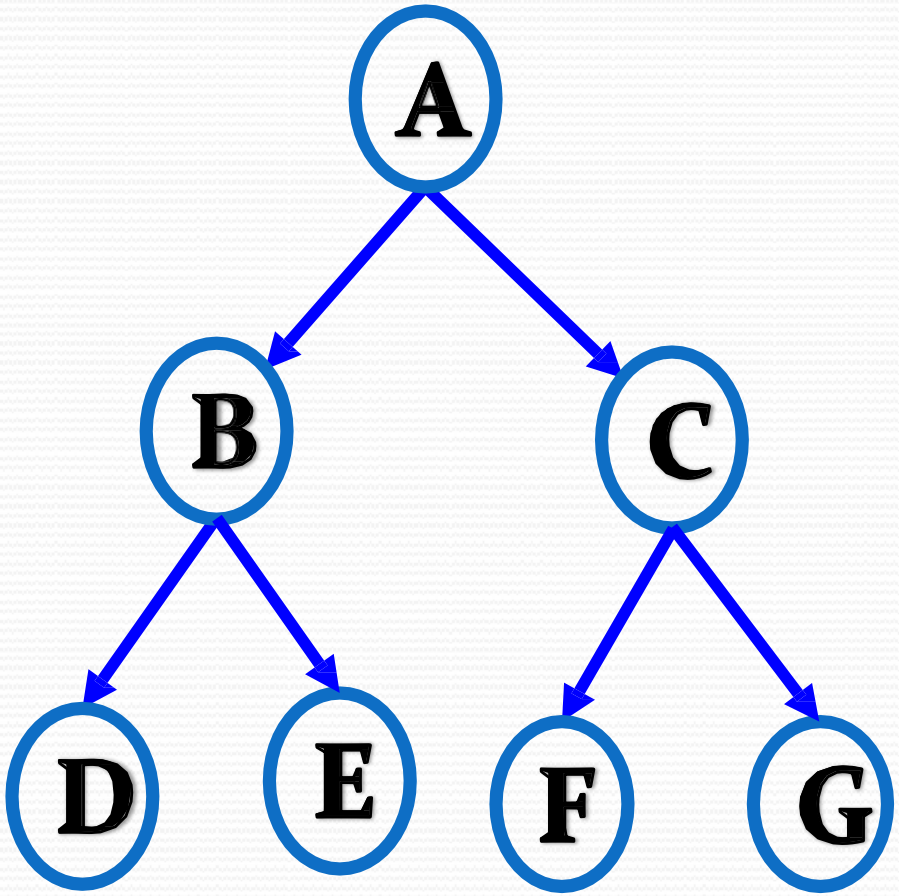
Inorder Traversal

1	10	8	4	6	3	2	5



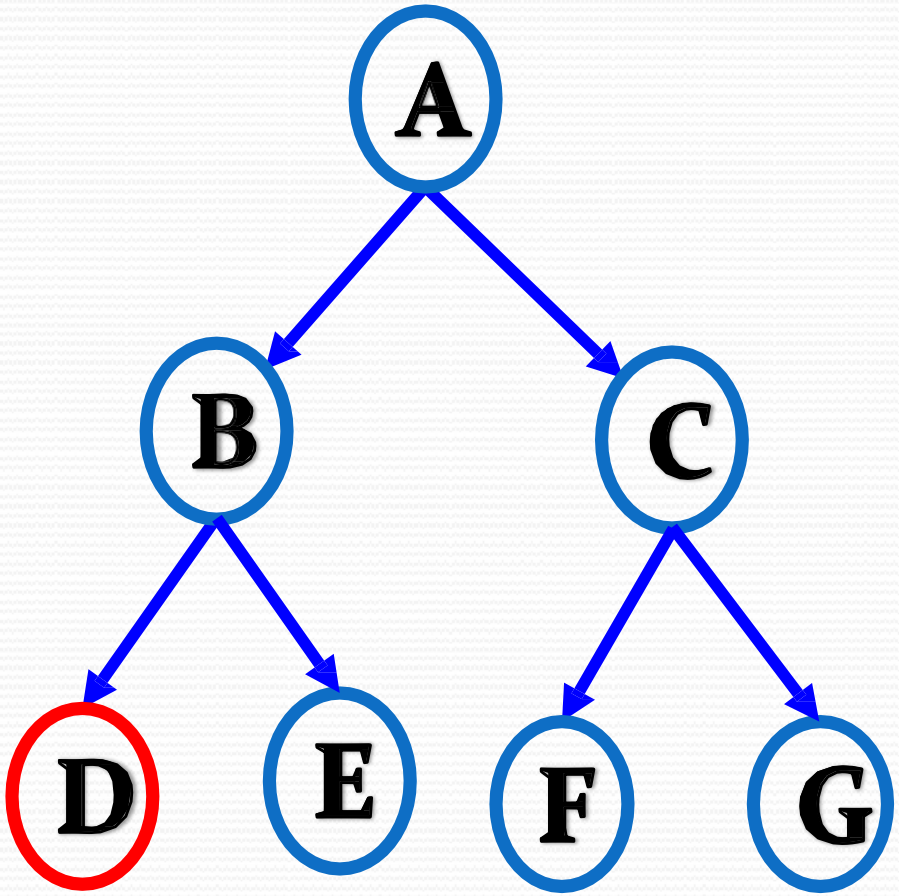
1	2	3	4	5	6	8	10

Tree Traversal : Post order



- Post order (L R V)

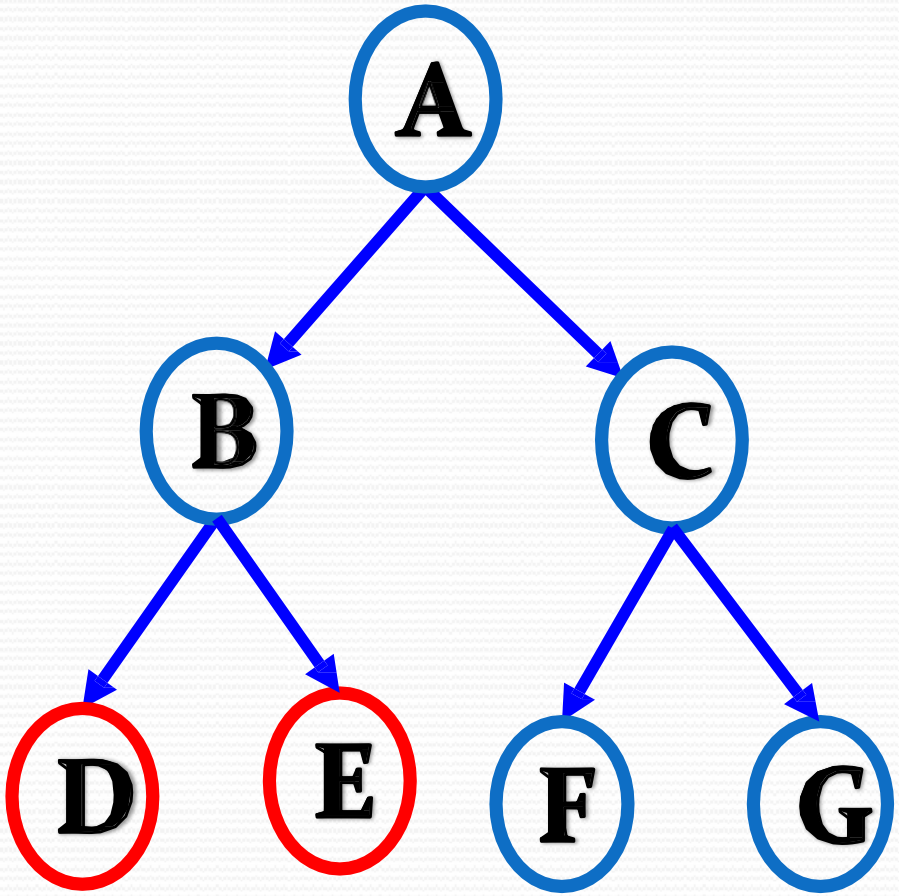
Tree Traversal : Post order



- Post order (L R V)

  D

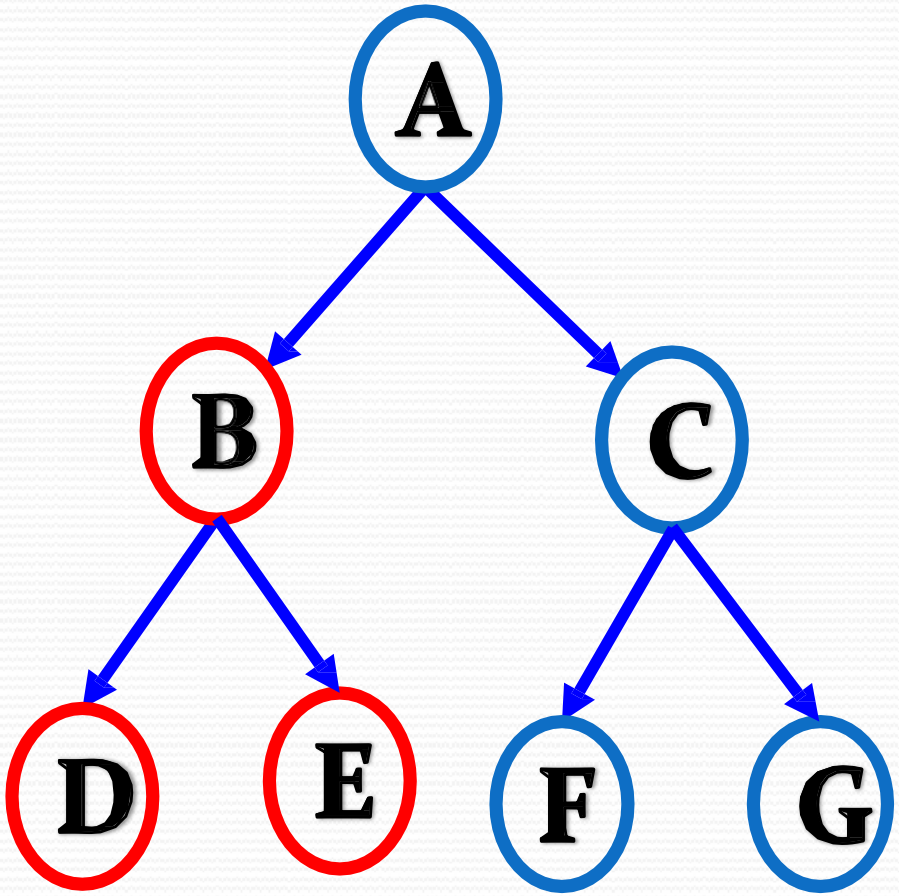
Tree Traversal : Post order



- Post order (L R V)

  D, E

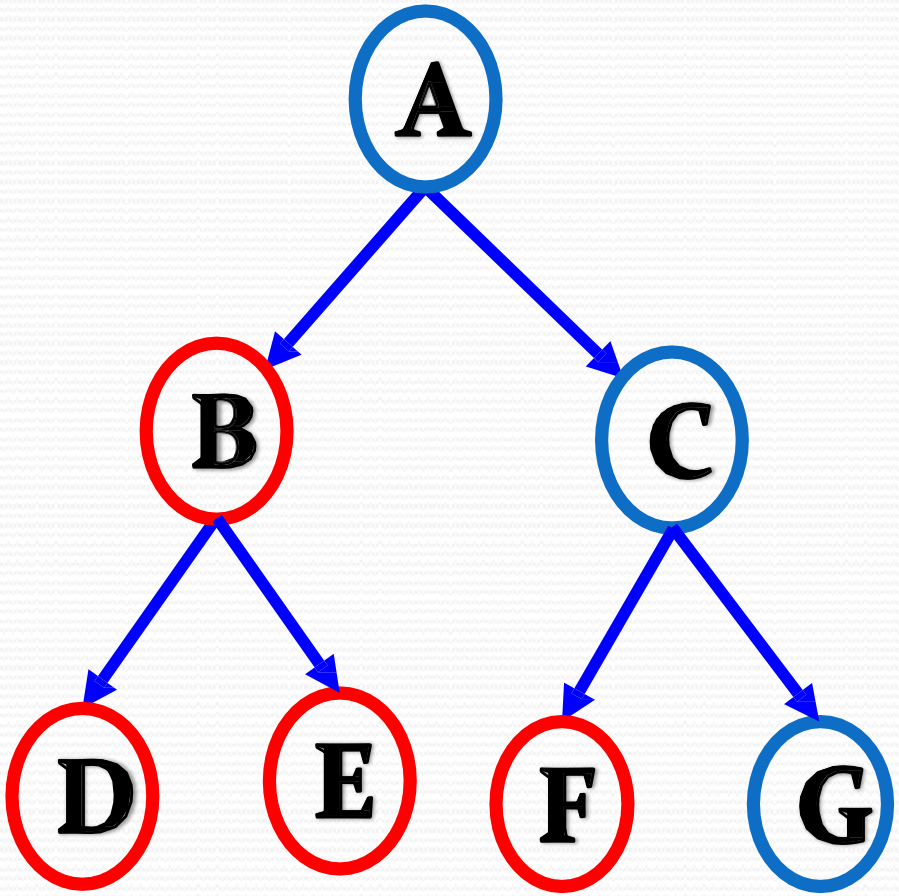
Tree Traversal : Post order



- Post order (L R V)

  D, E, B

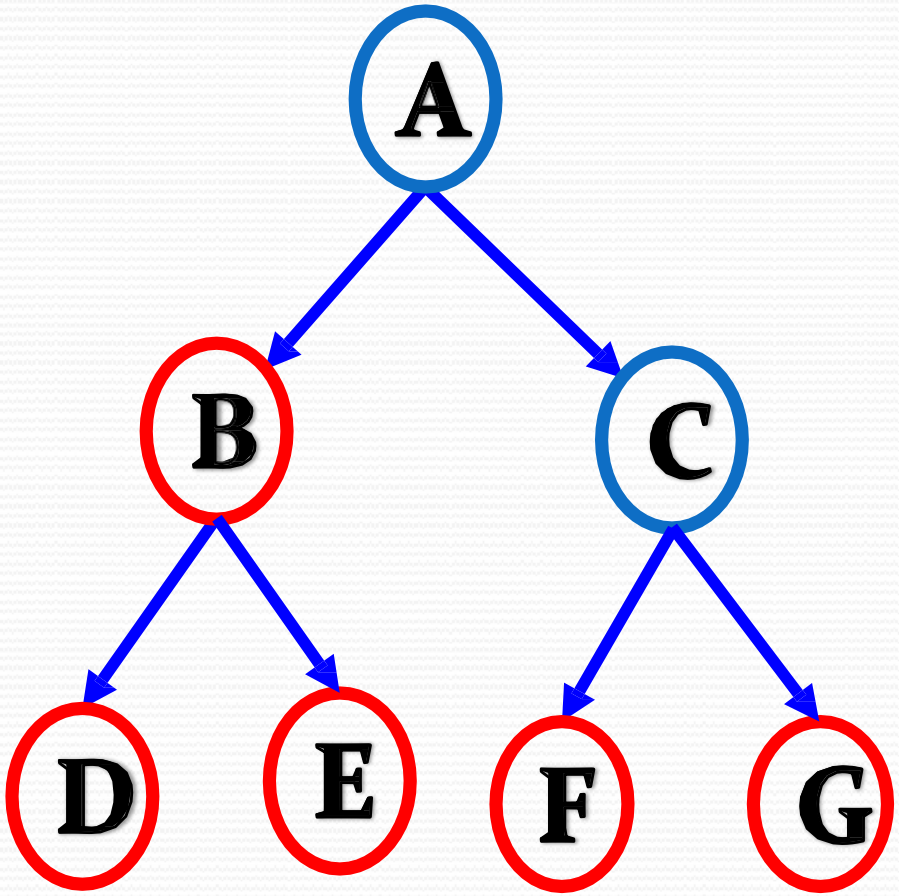
Tree Traversal : Post order



- Post order (L R V)

  D, E, B, F

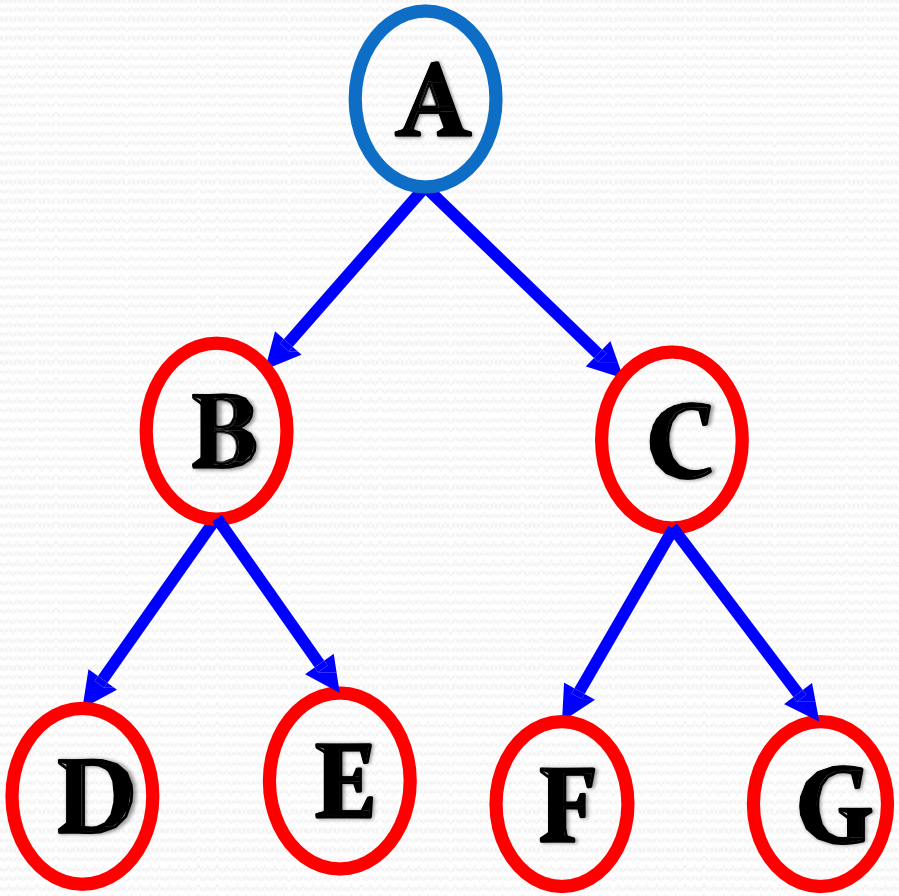
Tree Traversal : Post order



- Post order (L R V)

  D, E, B, F, G

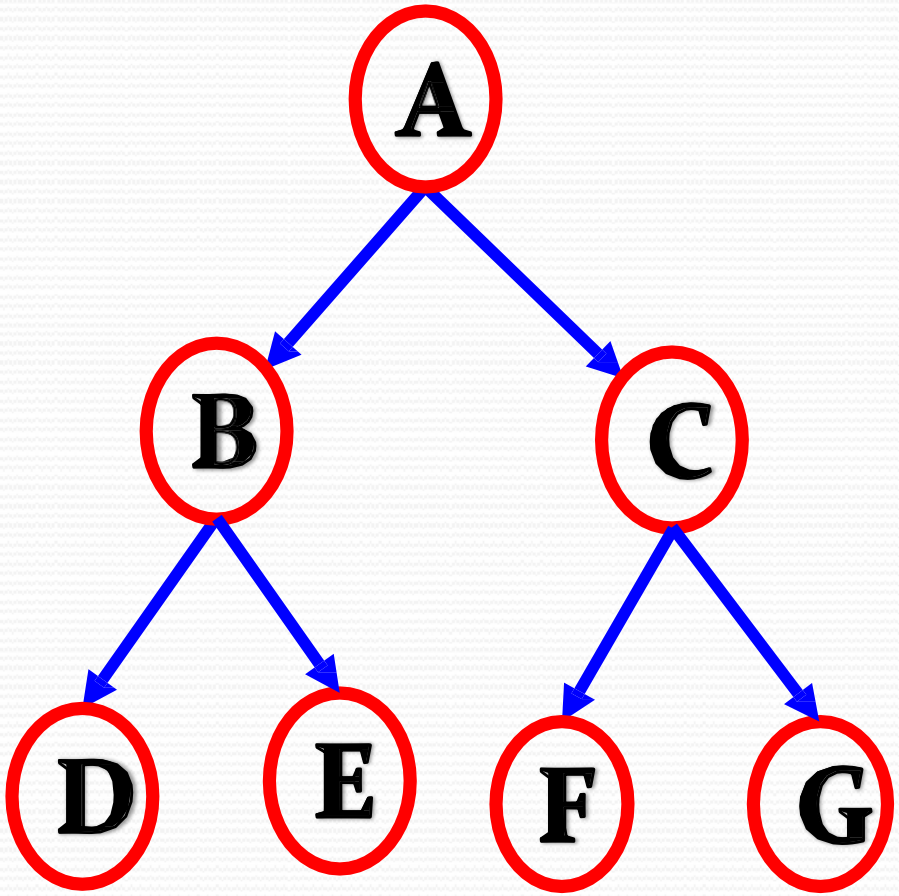
Tree Traversal : Post order



- Post order (L R V)

  D, E, B, F, G, C

Tree Traversal : Post order



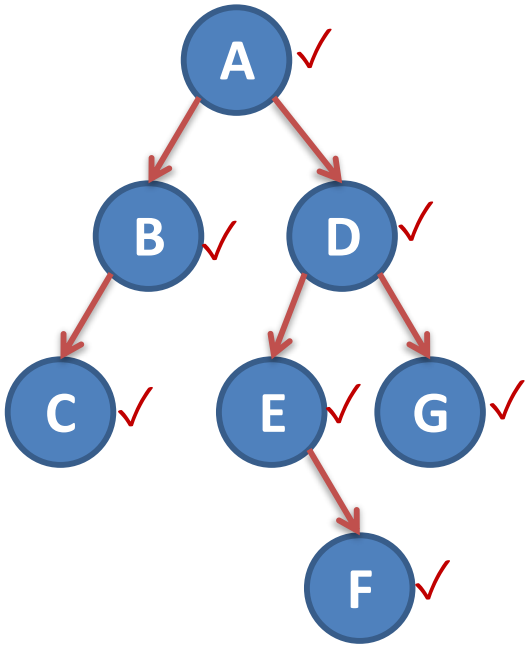
- Post order (L R V)

  D, E, B, F, G, C, A

Postorder Traversal

Postorder traversal of a binary tree is defined as follow

1. **Traverse** the **left subtree** in Postorder
2. **Traverse** the **right subtree** in Postorder
3. **Process** the **root node**



Postorder traversal of a given tree as

C B F E G D A

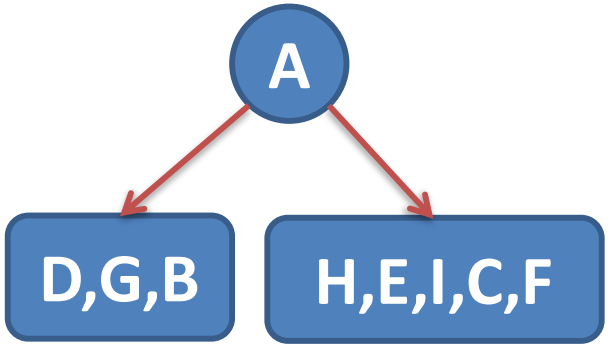
Construct Binary Tree from Traversal

Construct a Binary tree from the given **Inorder** and **Postorder** traversals

Inorder : D G B **A** H E I C F
Postorder : G D B H I E F C **A**
	• Step 1: Find the root node
	  • Preoder Traversal – first node is root node
	  • Postoder Traversal last node is root node
	• Step 2: Find Left & Right Sub Tree
	  • Inorder traversal gives Left and right sub tree

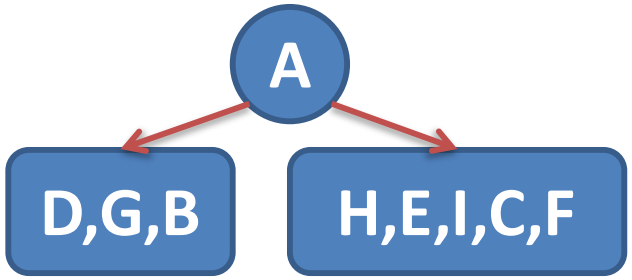
Postorder : G D B H I E F C **A**

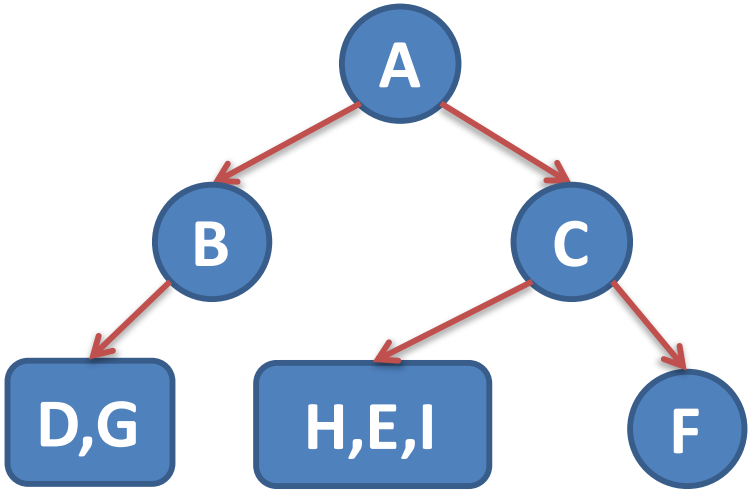
Inorder : D G B **A** H E I C F

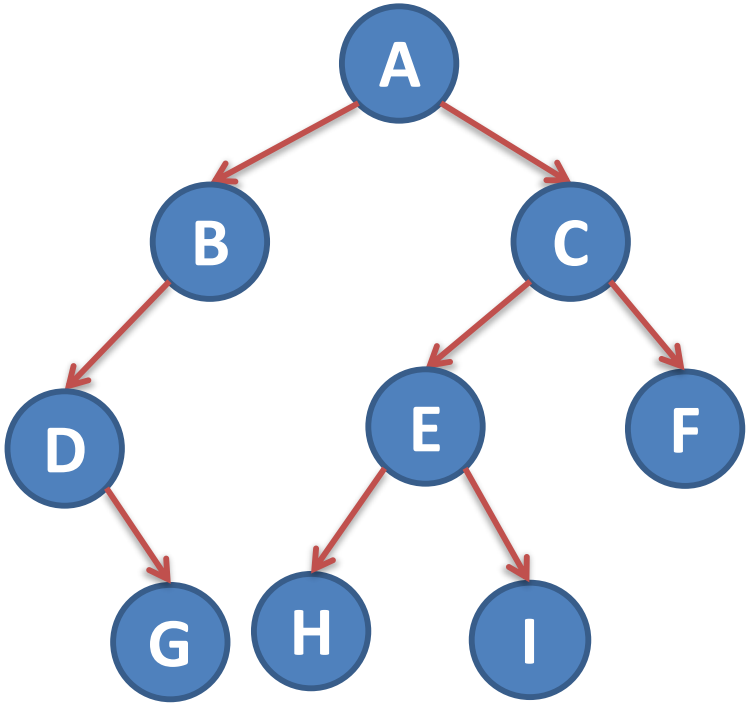


Construct Binary Tree from Traversal

Postorder : G D B H I E F C A | **Inorder :** D G B A H E I C F



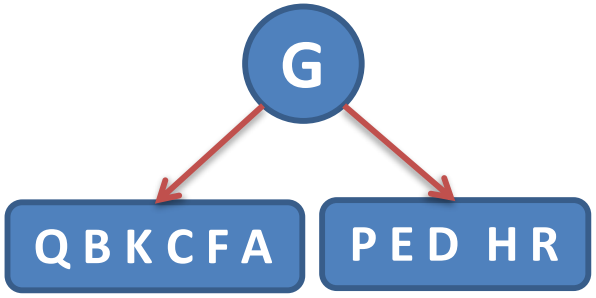


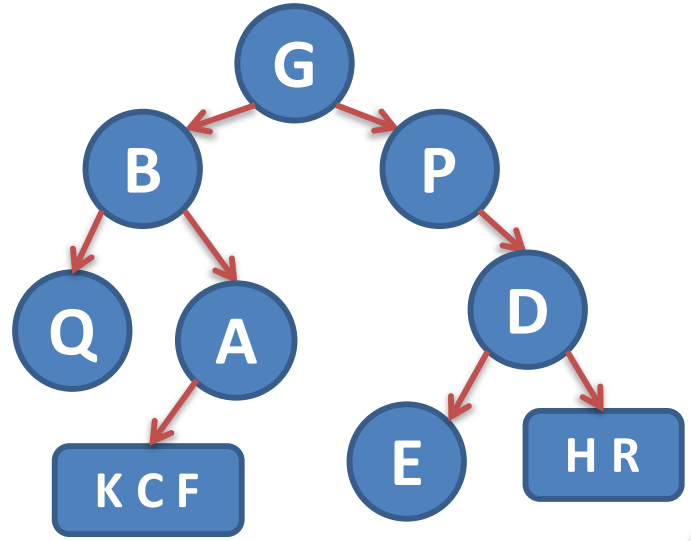


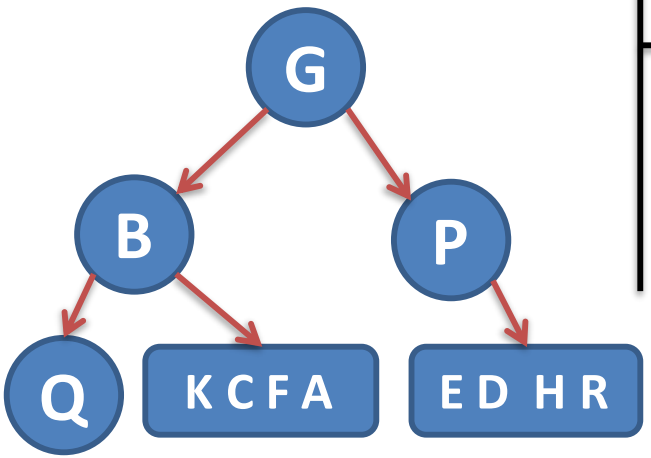
Construct Binary Tree from Traversal

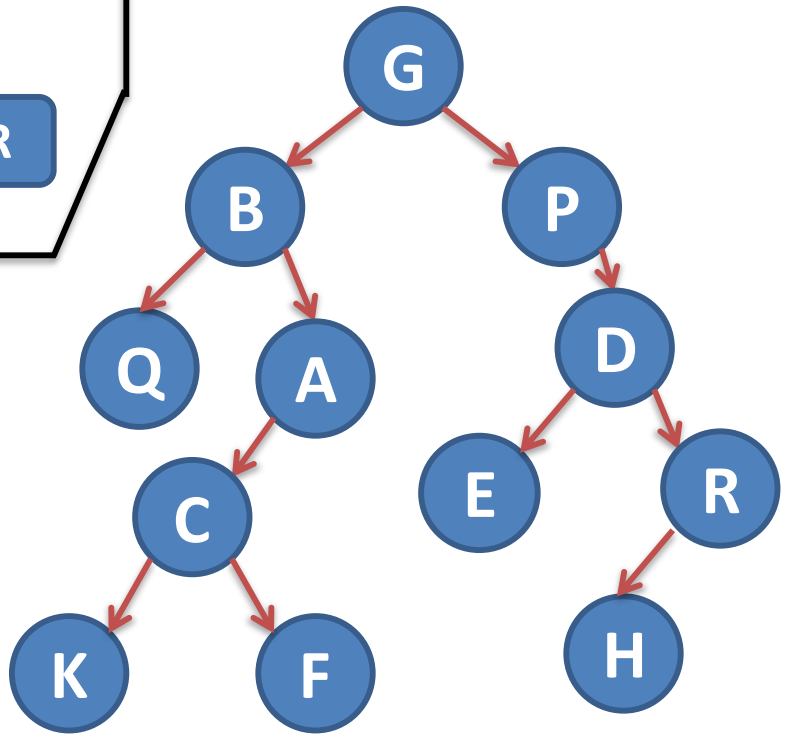
Preorder : G B Q A C K F P D E R H

Inorder : Q B K C F A G P E D H R







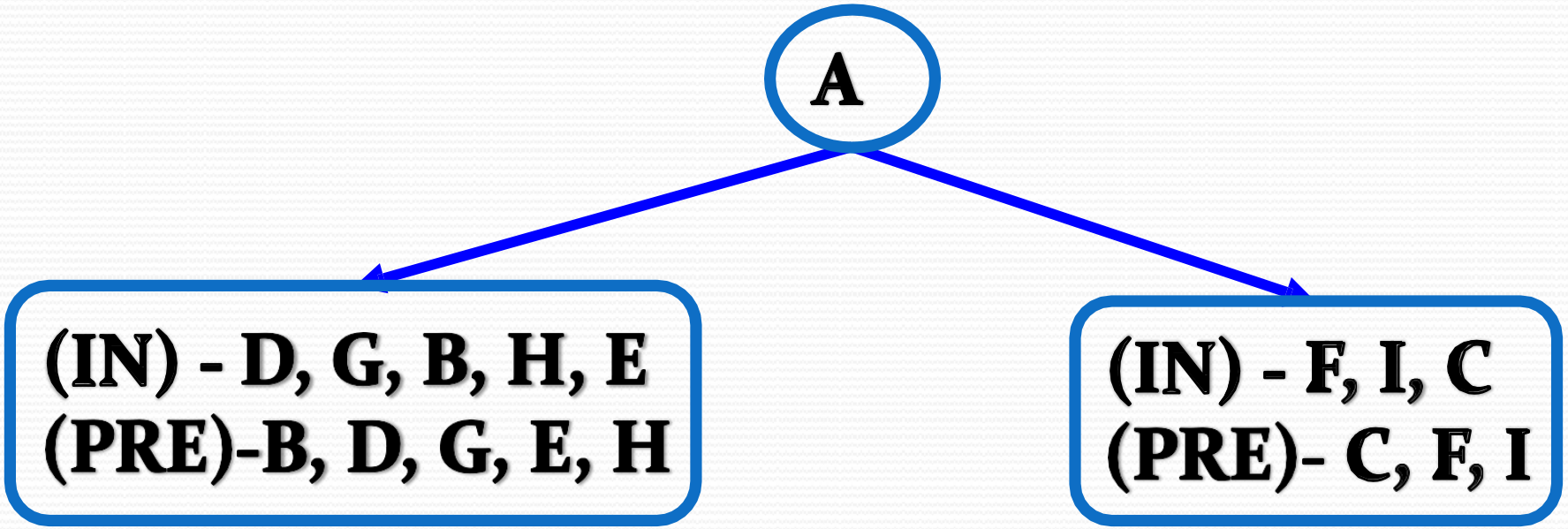


Constructing Binary Tree

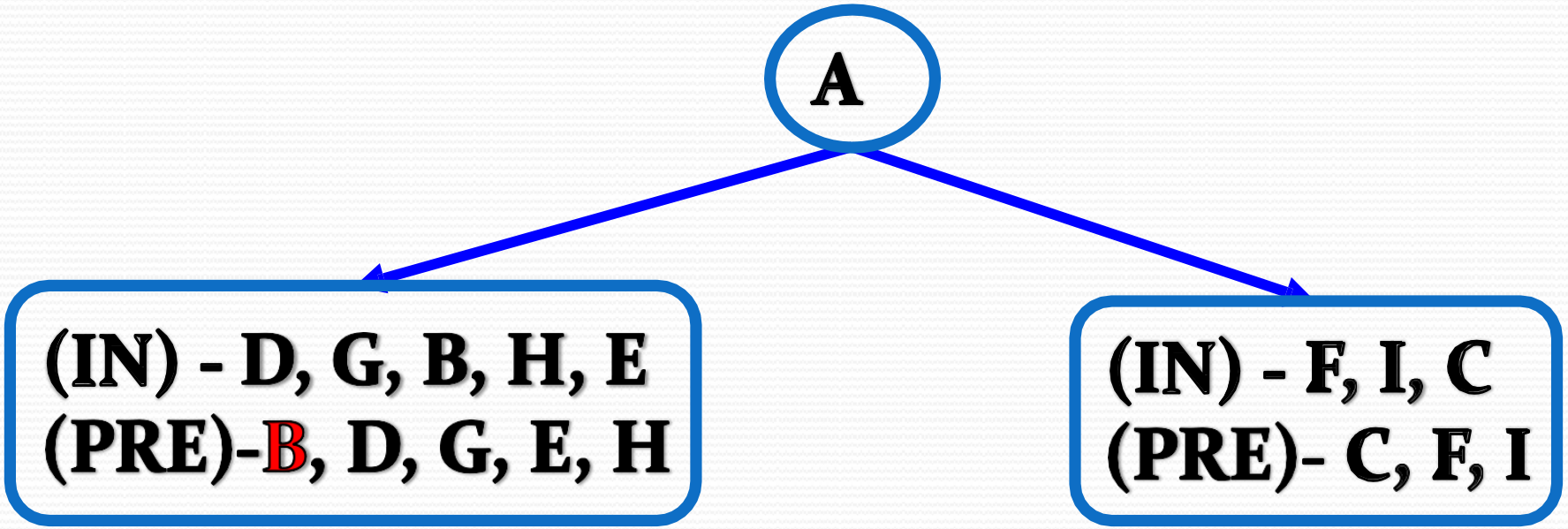
- In order – D, G, B, H, E, A, F, I, C
- Pre order – A, B, D, G, E, H, C, F, I

- Step 1 : finding the root
  - Root Node – A (from preorder)
- Step 2 : Find left and right part of the root
  - Left part - **(IN) - D, G, B, H, E**
    (PRE)-B, D, G, E, H
  - Right part - **(IN) - F, I, C**
    (PRE)- C, F, I

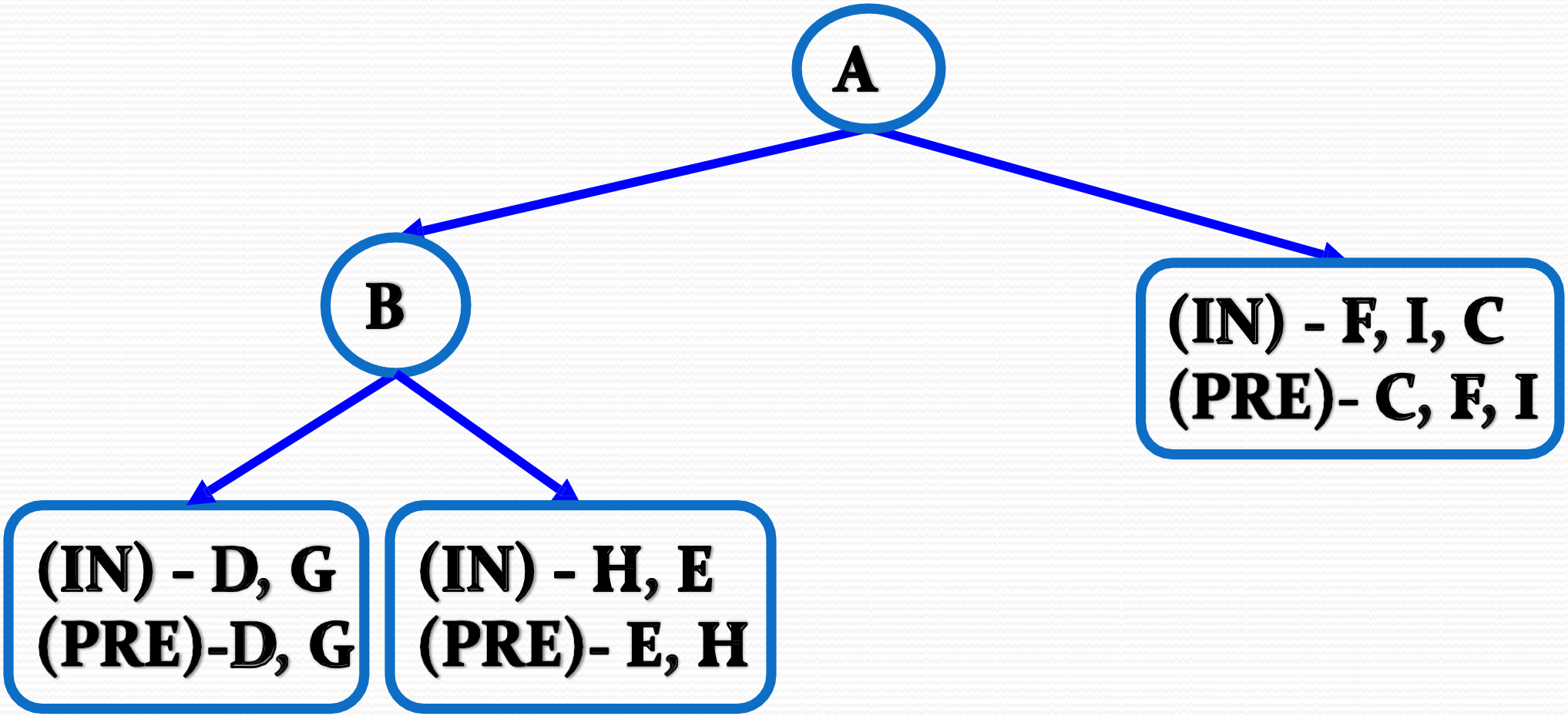
Constructing Binary Tree



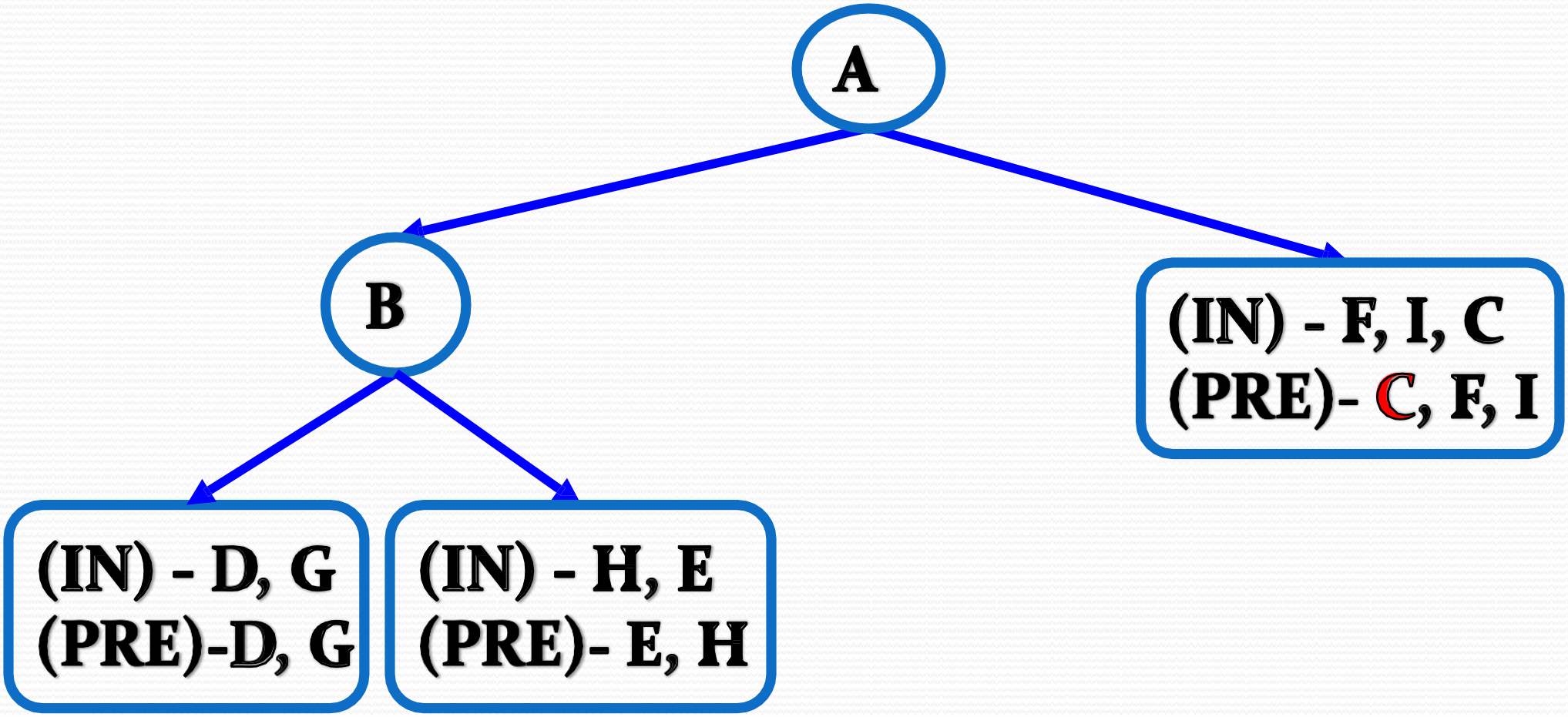
Constructing Binary Tree



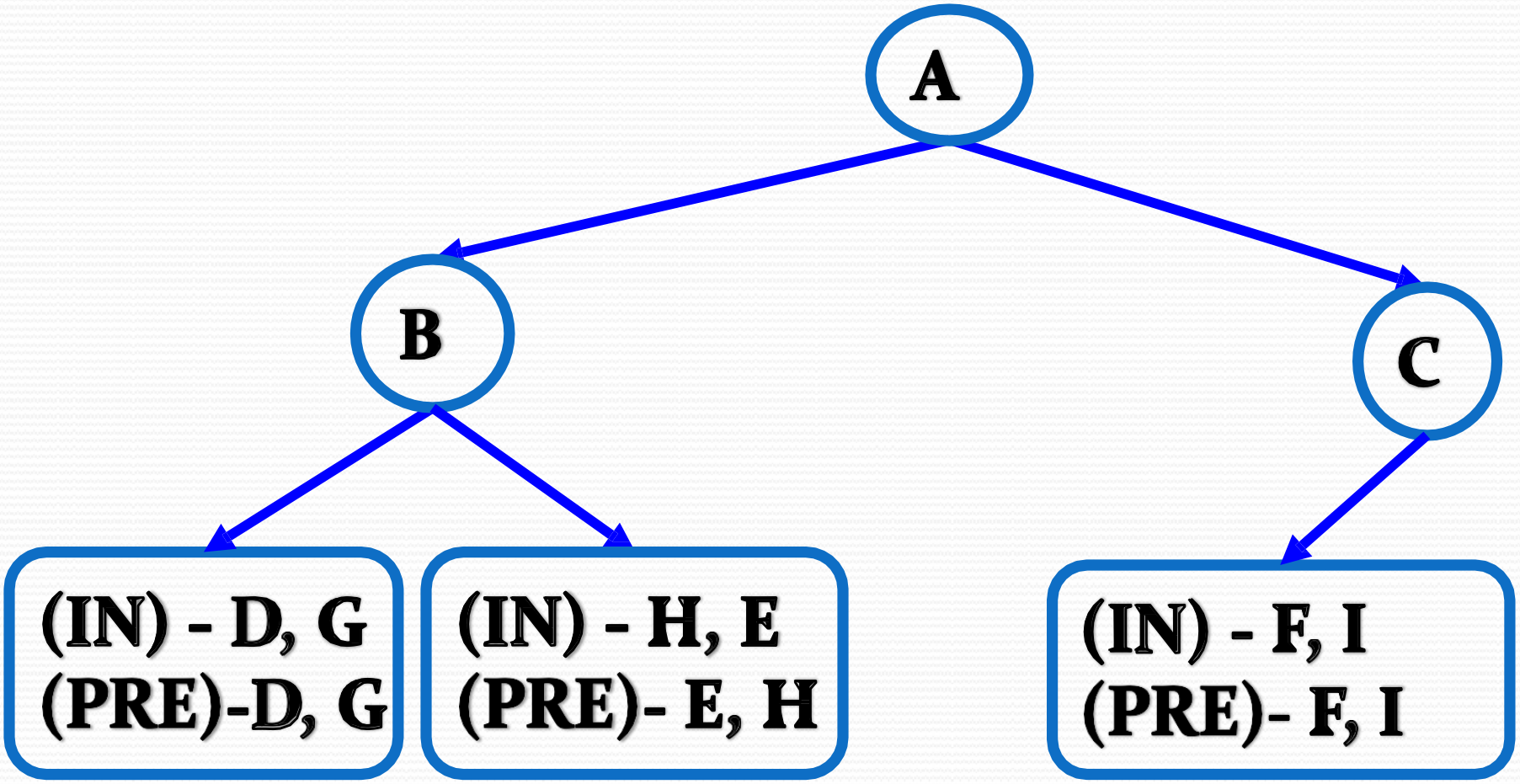
Constructing Binary Tree



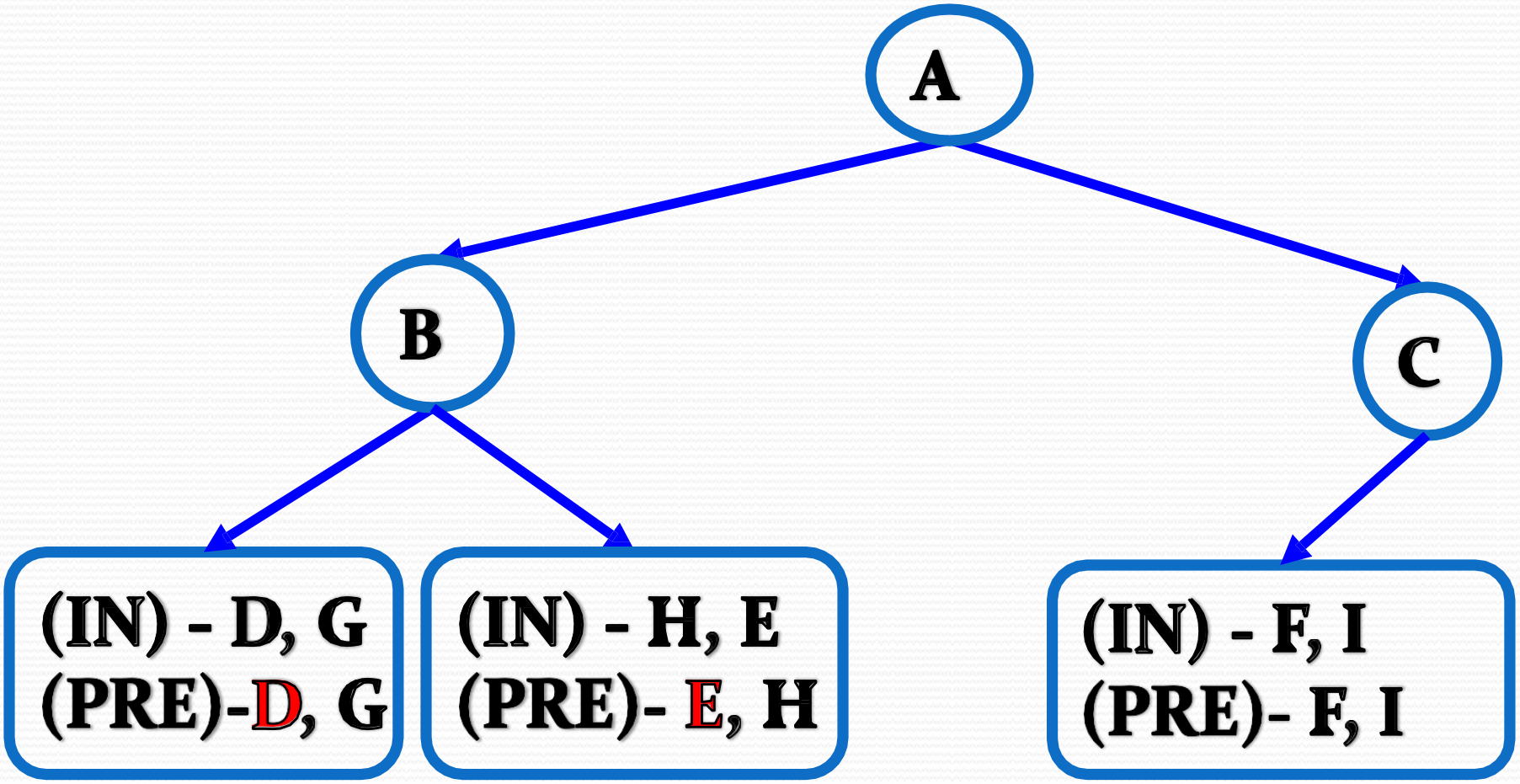
Constructing Binary Tree



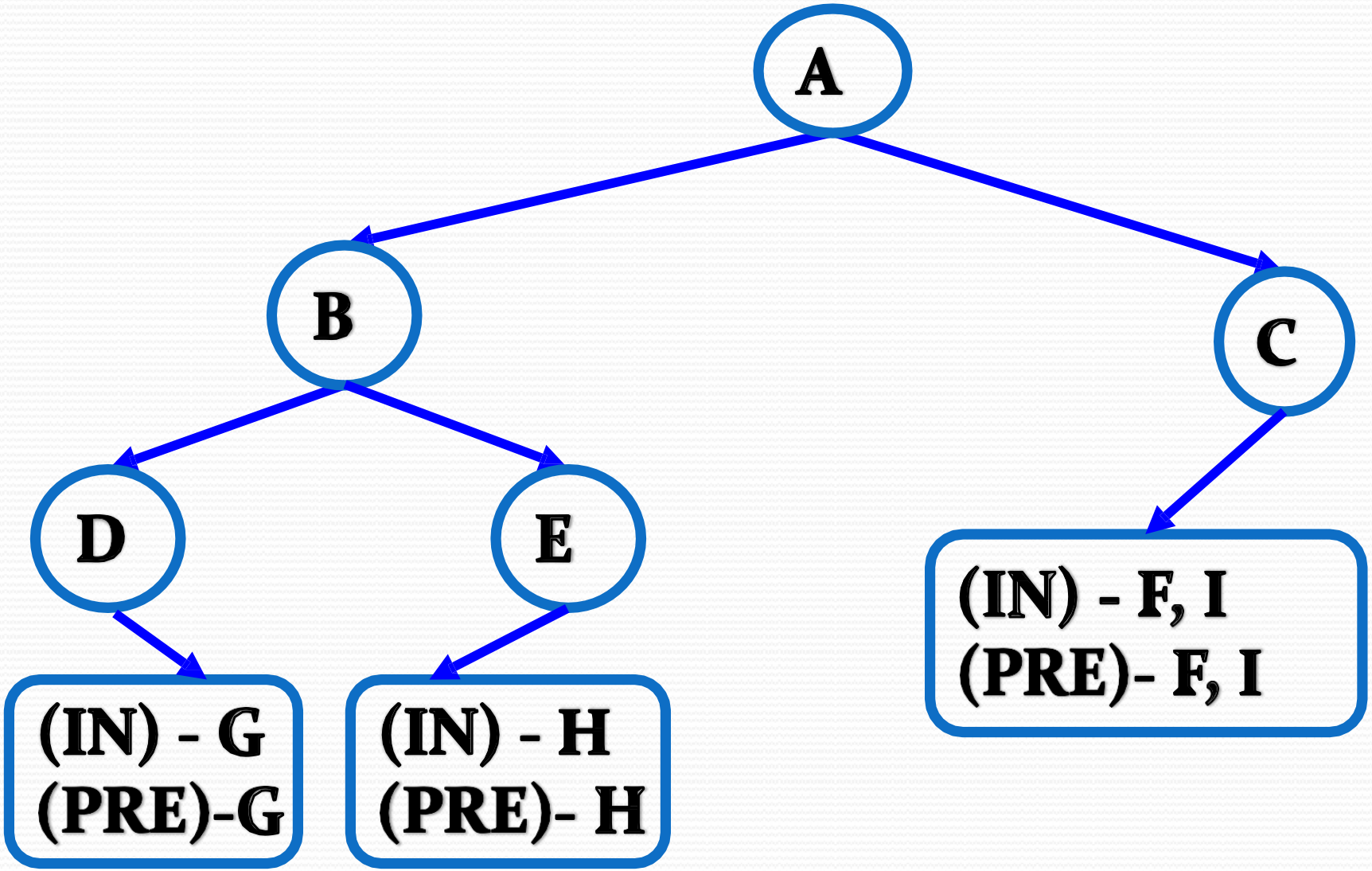
Constructing Binary Tree



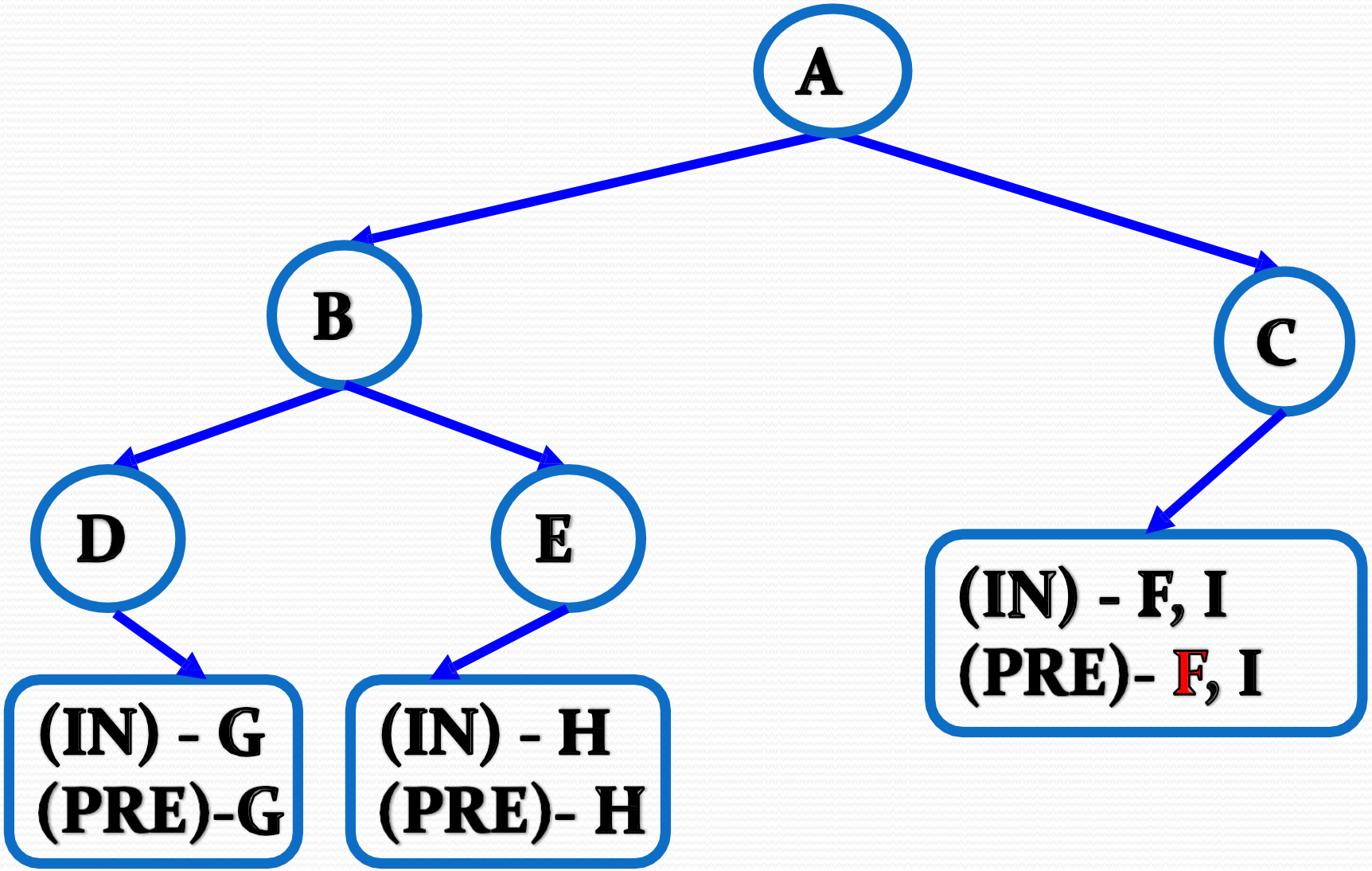
Constructing Binary Tree



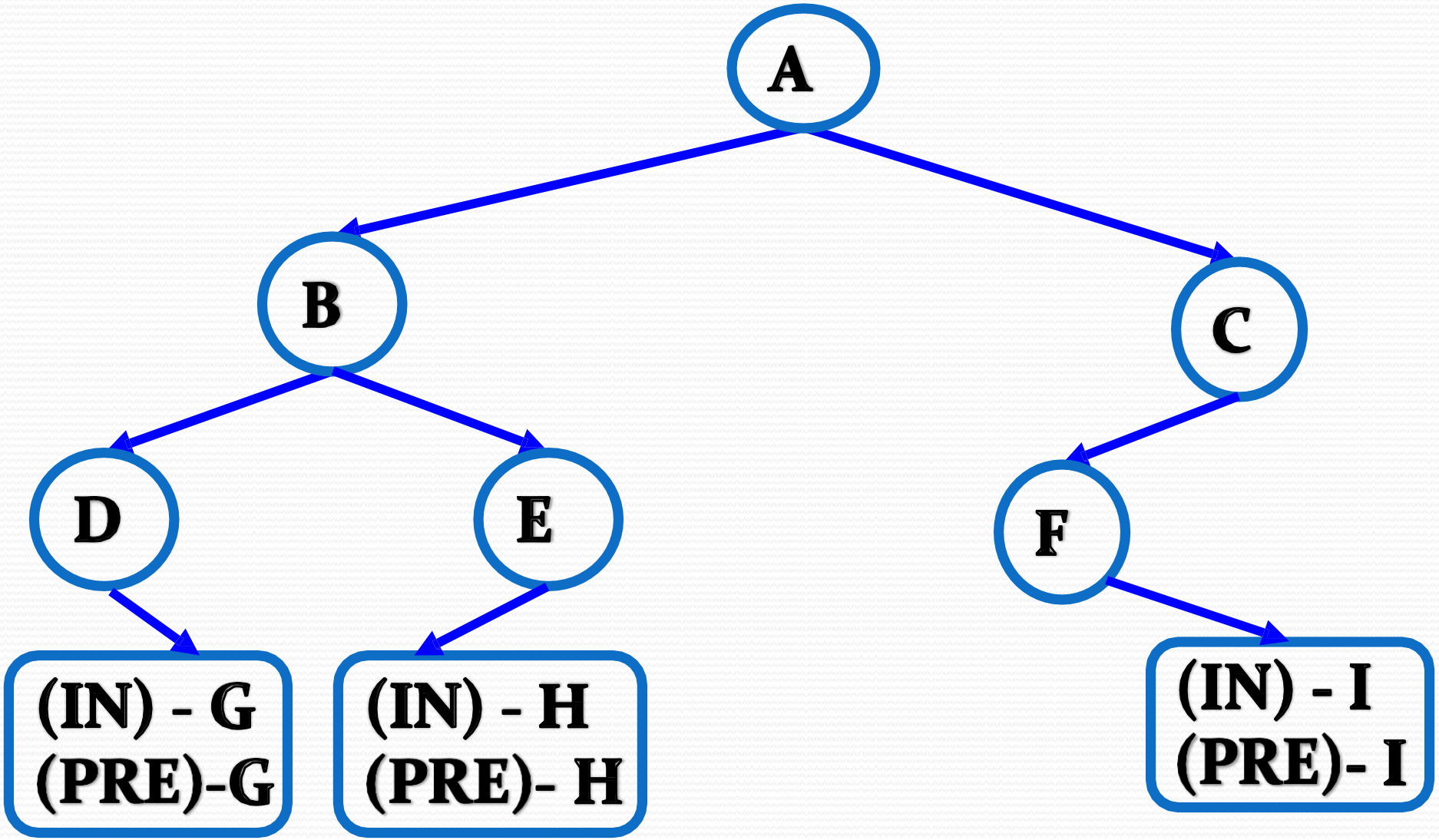
Constructing Binary Tree



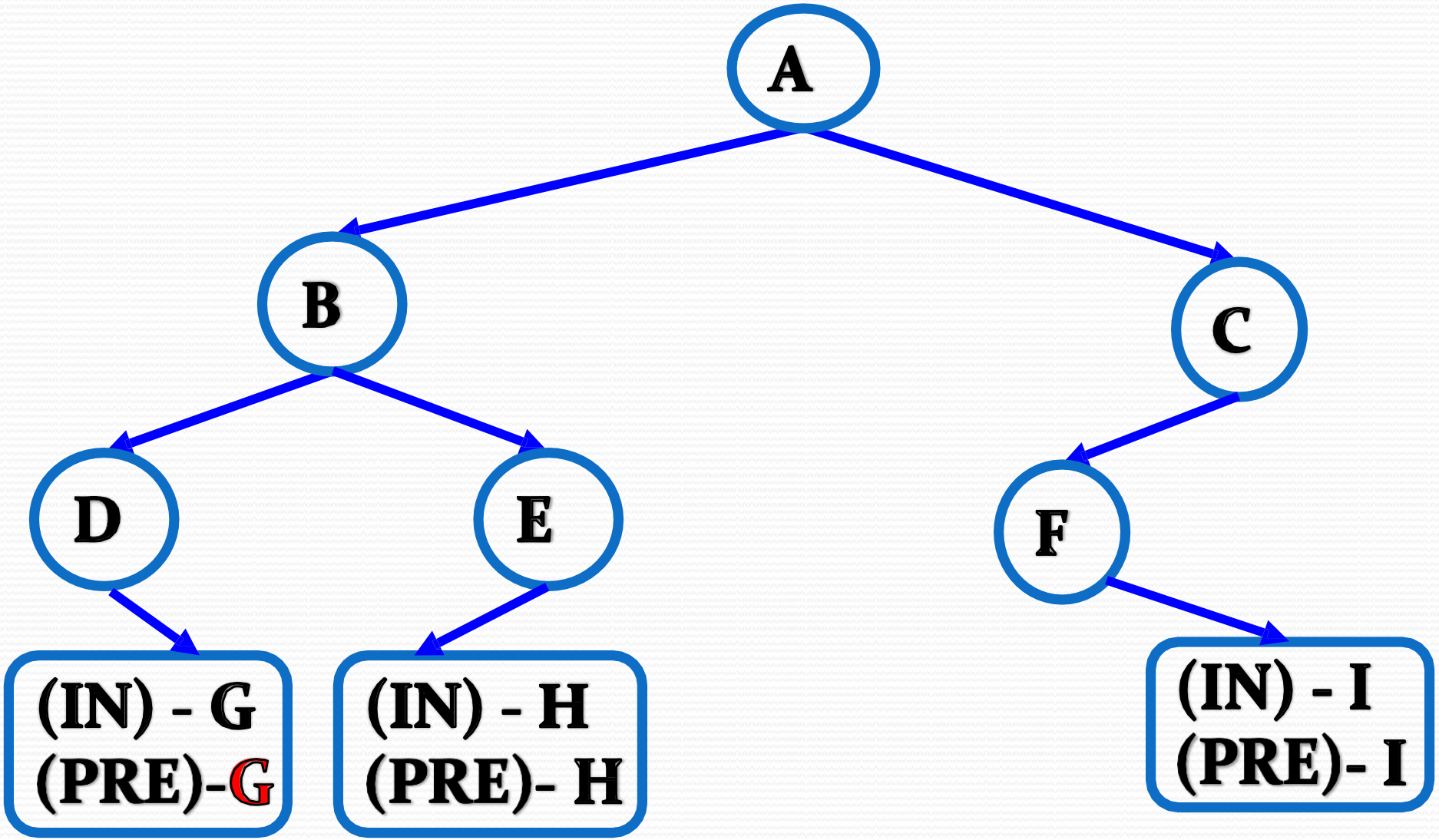
Constructing Binary Tree



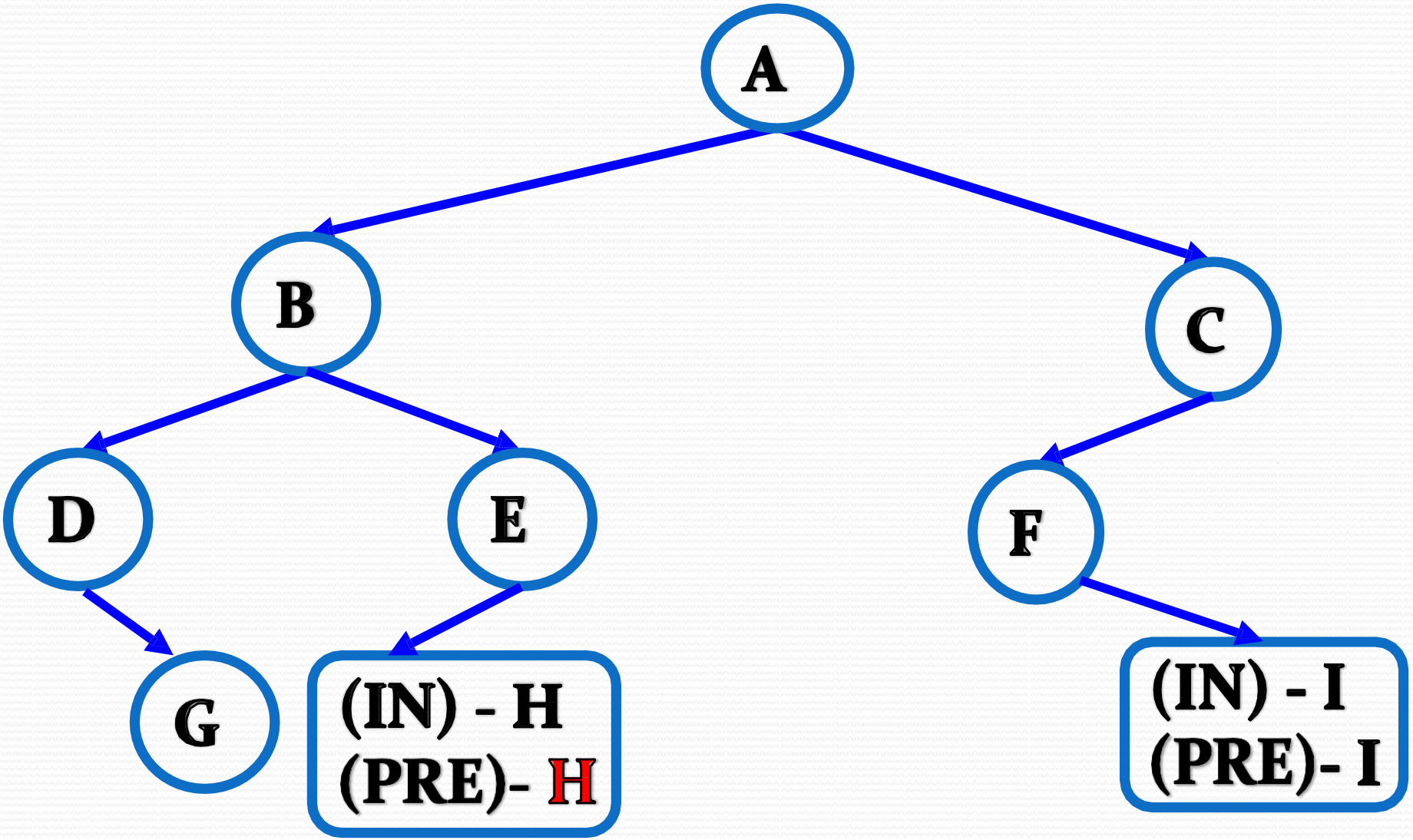
Constructing Binary Tree



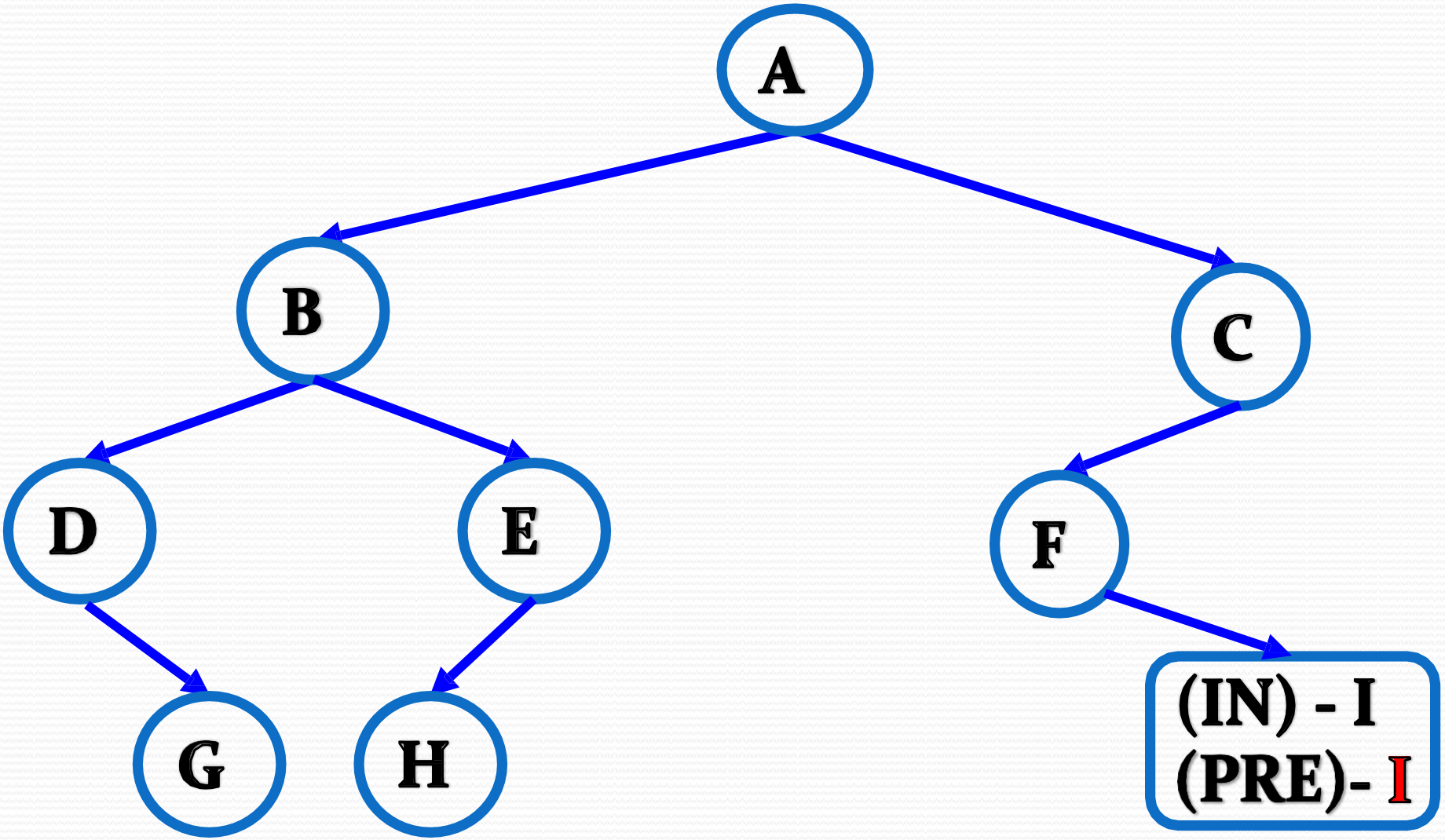
Constructing Binary Tree



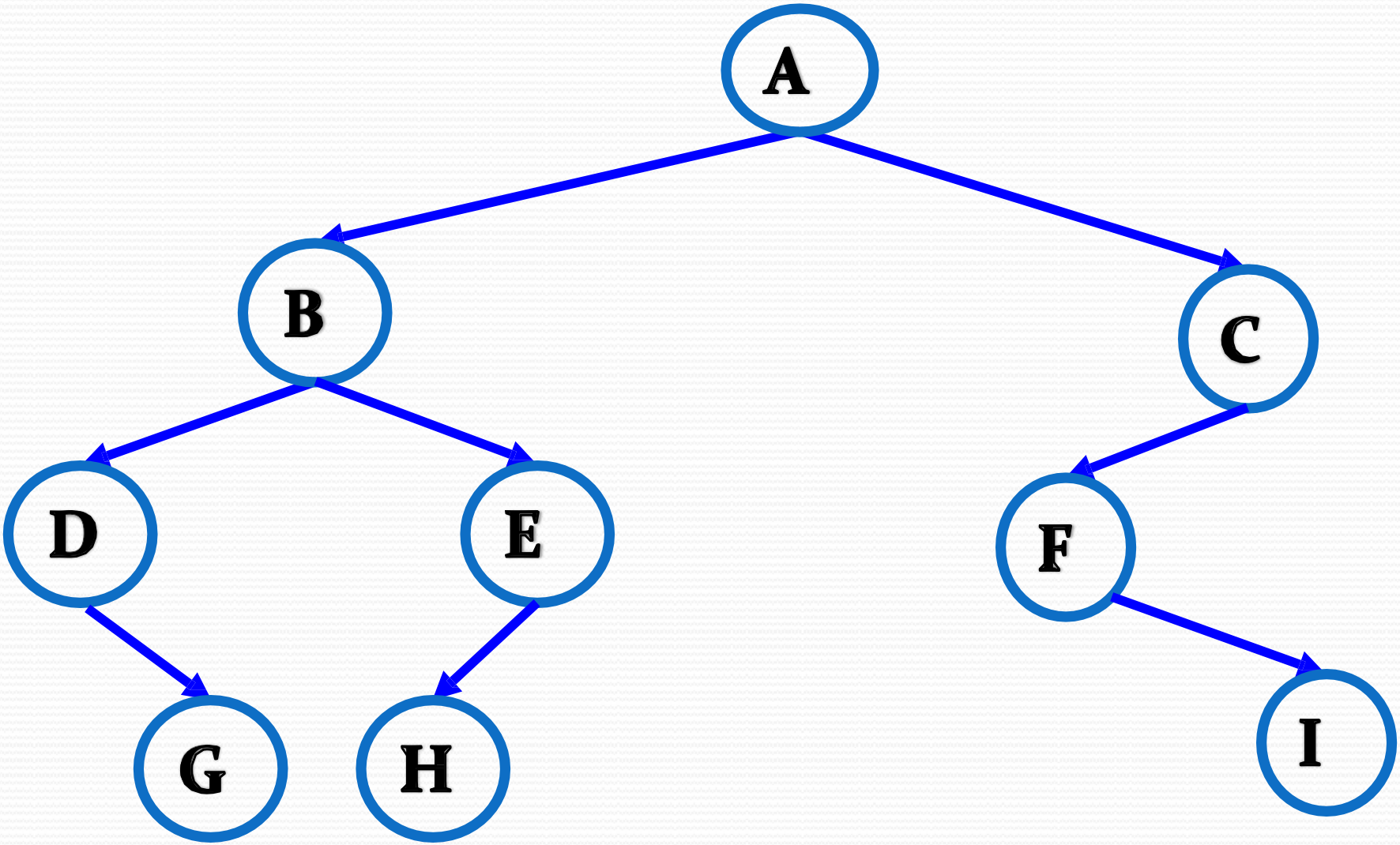
Constructing Binary Tree



Constructing Binary Tree



Constructing Binary Tree



Binary Search Tree

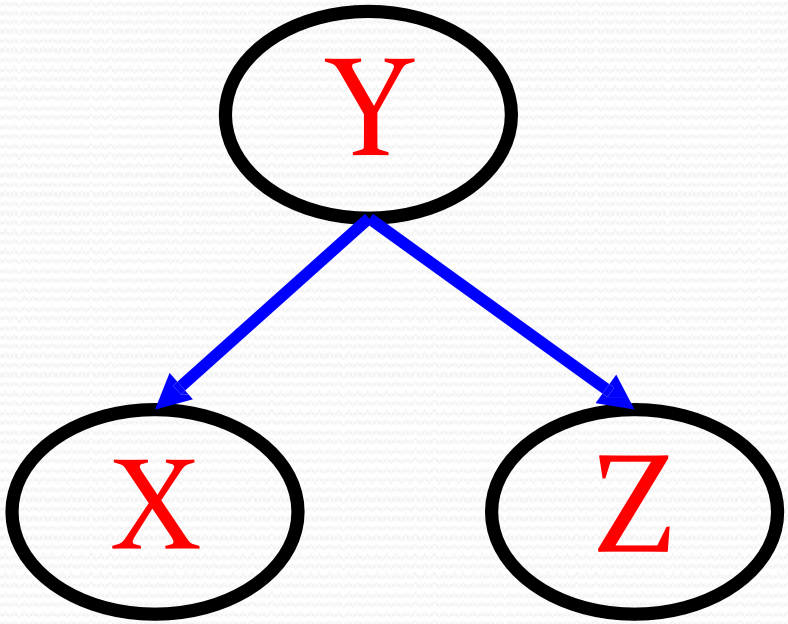
Binary Search Tree

The value at any node,

- Greater than every value in left subtree
- Smaller than every value in right subtree

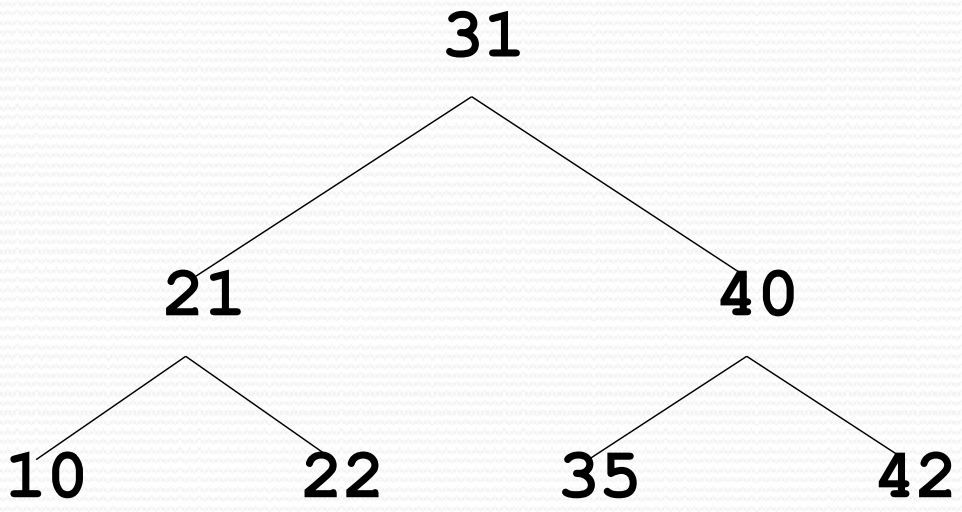
– Example

- Y > X
- Y < Z



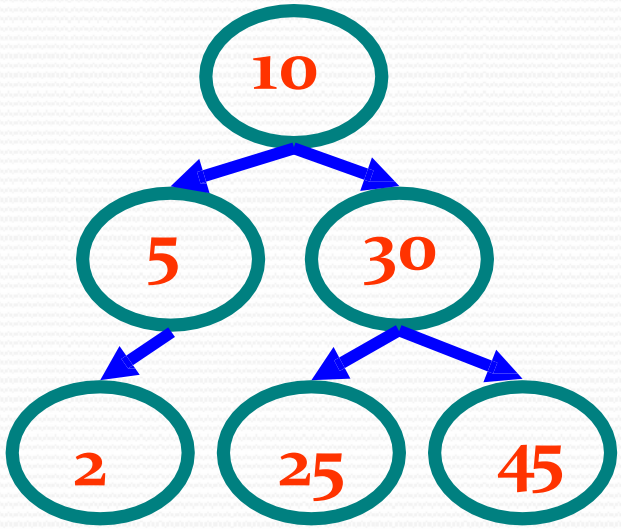
Binary Search Tree

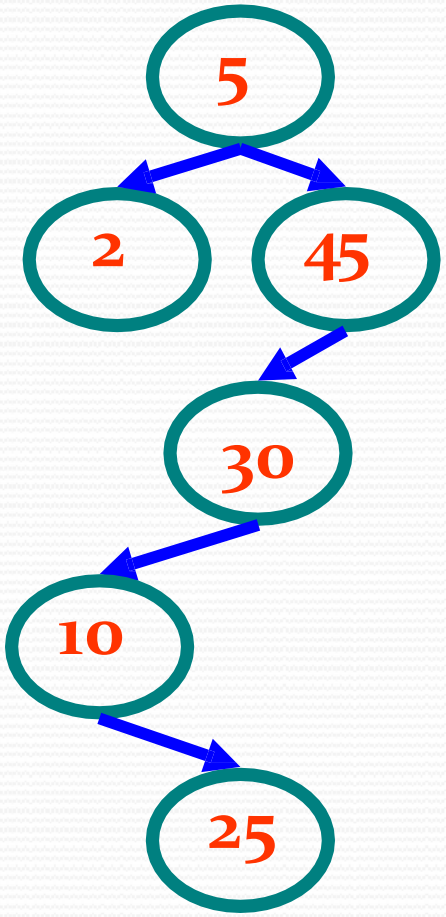
- Values in left sub tree less than parent
- Values in right sub tree greater than parent
- Fast searches in a Binary Search tree, maximum of log n comparisons



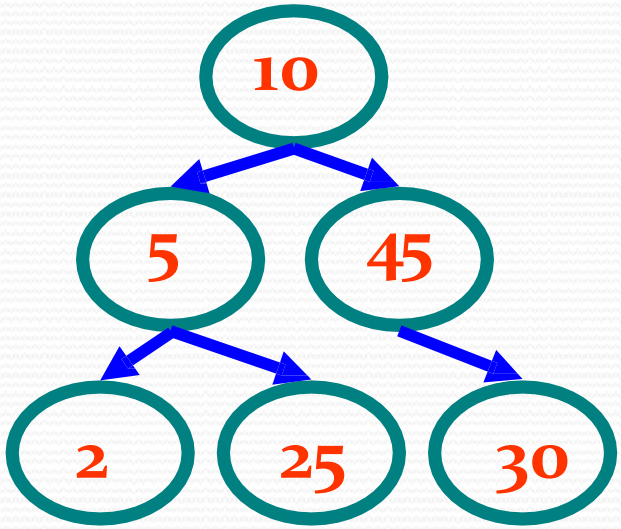
Binary Search Trees

- Examples





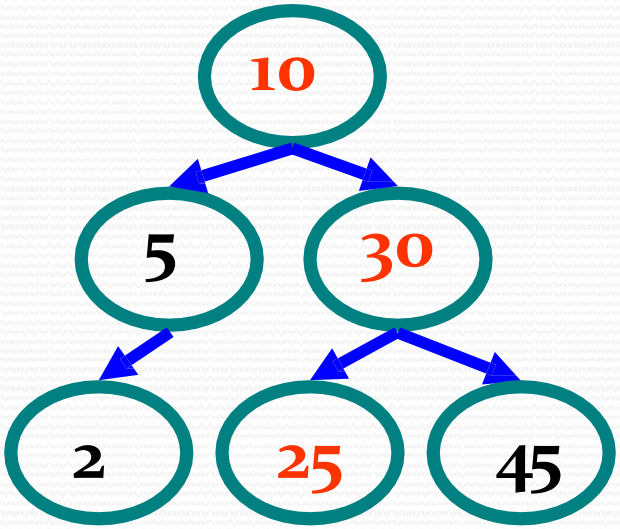
Binary search trees



Not a binary search tree

Example Binary Searches

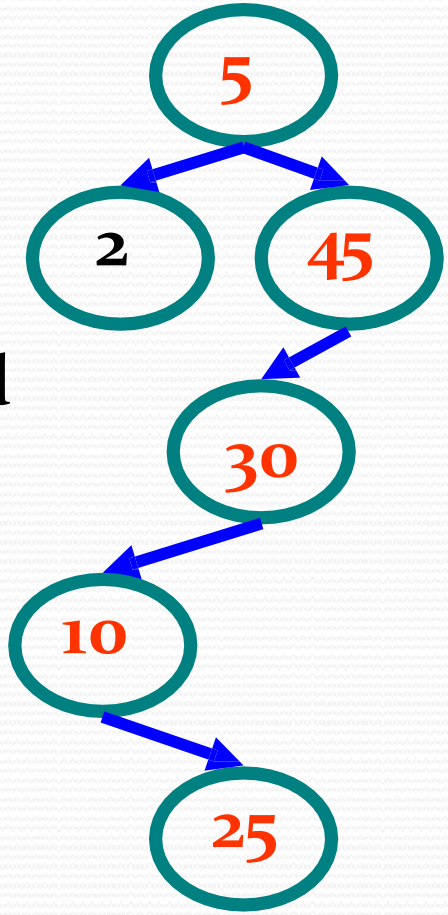
- search (root, 25)



10 < 25, right

30 > 25, left

25 = 25, found



5 < 25, right

45 > 25, left

30 > 25, left

10 < 25, right

25 = 25, found

Algorithm for Binary Search Tree

- A) compare ITEM with the root node N of the tree
  - i) if ITEM<N, proceed to the left child of N.
  - ii) if ITEM>N, proceed to the right child of N.
- B) repeat step (A) until one of the following occurs
  - i) we meet a node N such that ITEM=N, i.e. search is successful.
  - ii) we meet an empty sub tree, i.e. the search is unsuccessful.

Binary Tree Implementation

```
typedef struct node
{
    int data;
    struct node *lc,*rc;
};
```

Iterative Search of Binary Search Tree

```
search()
{
      while (n != NULL)
      {
      if (n->data == item)       // Found it
            return n;
         if (n->data > item)              // In left subtree
            n = n->lc;
         else                        // In right subtree
            n = n->rc;
      }
      return null;
}
```

Recursive Search of Binary Search Tree

```
Search(node *n, info)
{
    if (n == NULL)                  // Not found
        return( n );
    else if (n->data == item)       // Found it
        return( n );
    else if (n->data > item)        // In left subtree
        return search( n->left, item );
    else                            // In right subtree
        return search( n->right, item );
}
```

Recursive Search of Binary Search Tree

```
searchElement (TREE, VAL)

Step 1: IF TREE -> DATA = VAL OR TREE = NULL
          Return TREE
        ELSE
         IF VAL < TREE -> DATA
           Return searchElement(TREE -> LEFT, VAL)
         ELSE
           Return searchElement(TREE -> RIGHT, VAL)
         [END OF IF]
        [END OF IF]
Step 2: END
```

Insertion in a Binary Search Tree

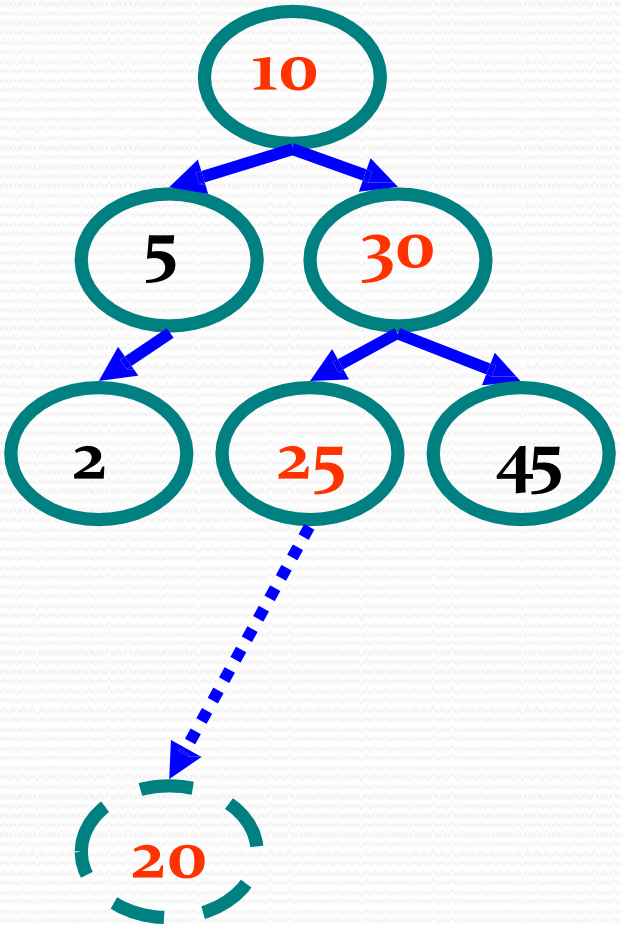
- Algorithm
  1. Perform search for value X
  2. Search will end at node Y (if X not in tree)
  3. If X < Y, insert new leaf X as new left subtree for Y
  4. If X > Y, insert new leaf X as new right subtree for Y

```
Insert (TREE, VAL)

Step 1: IF TREE = NULL
            Allocate memory for TREE
            SET TREE -> DATA = VAL
            SET TREE -> LEFT = TREE -> RIGHT = NULL
        ELSE
            IF VAL < TREE -> DATA
                  Insert(TREE -> LEFT, VAL)
            ELSE
                  Insert(TREE -> RIGHT, VAL)
            [END OF IF]
        [END OF IF]
Step 2: END
```

Insertion in a Binary Search Tree

- Insert (20)



10 < 20, right

30 > 20, left

25 > 20, left

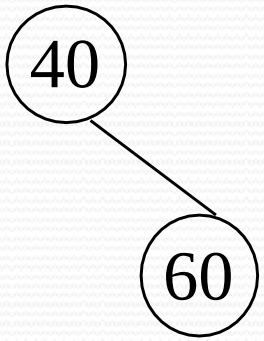
Insert 20 on left

Insertion in a Binary Tree

40, 60, 50, 33, 55, 11

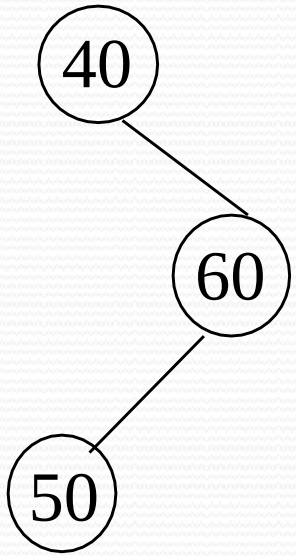
Insertion in a Binary Tree

40, 60, 50, 33, 55, 11



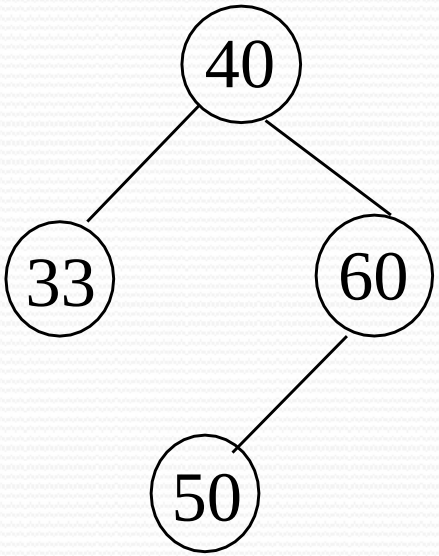
Insertion in a Binary Tree

40, 60, 50, 33, 55, 11



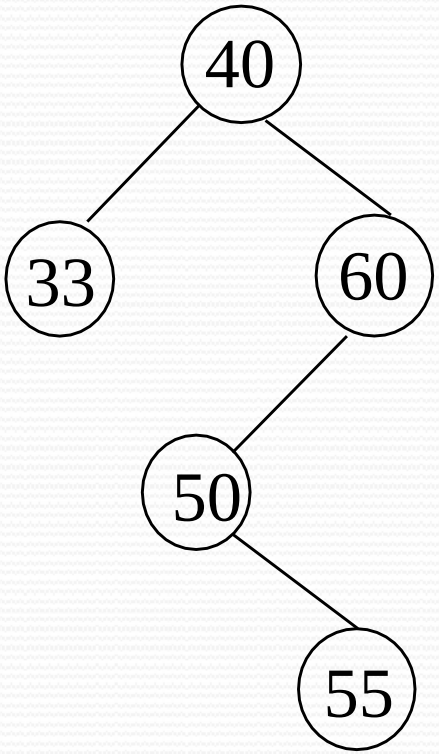
Insertion in a Binary Tree

40, 60, 50, 33, 55, 11



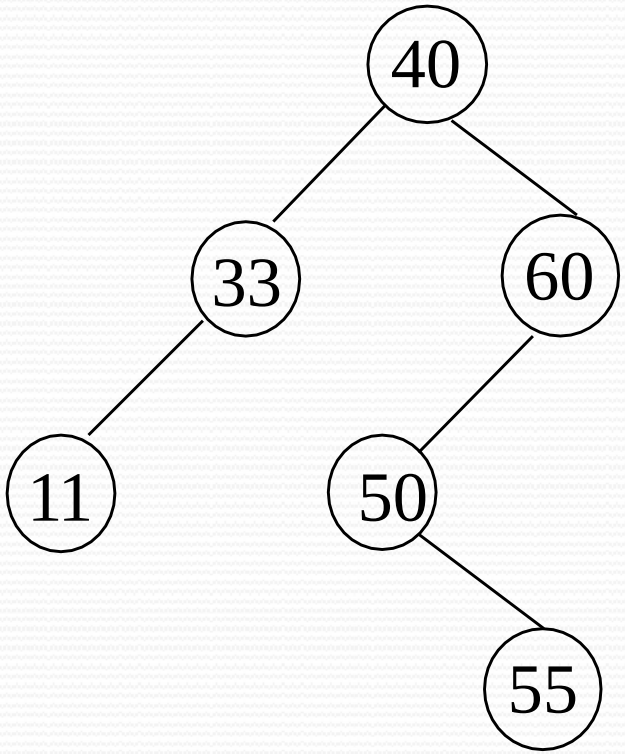
Insertion in a Binary Tree

40, 60, 50, 33, 55, 11



Insertion in a Binary Tree

40, 60, 50, 33, 55, 11



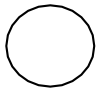
Binary Tree Insertion

Binary Tree Insertion

1	10	8	4	6	3	2	5

Binary Tree Insertion

1	10	8	4	6	3	2	5



Binary Tree Insertion

1	10	8	4	6	3	2	5

Binary Tree Insertion

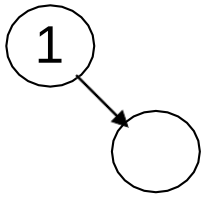
1	10	8	4	6	3	2	5

Binary Tree Insertion

1	10	8	4	6	3	2	5

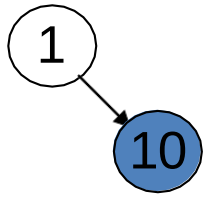
Binary Tree Insertion

1	10	8	4	6	3	2	5



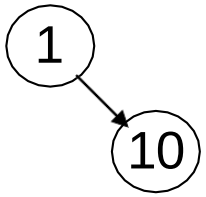
Binary Tree Insertion

1	10	8	4	6	3	2	5



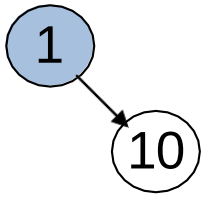
Binary Tree Insertion

1	10	8	4	6	3	2	5



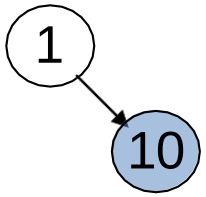
Binary Tree Insertion

1	10	8	4	6	3	2	5



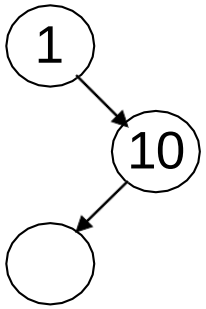
Binary Tree Insertion

1	10	8	4	6	3	2	5



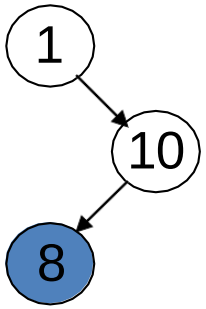
Binary Tree Insertion

1	10	8	4	6	3	2	5



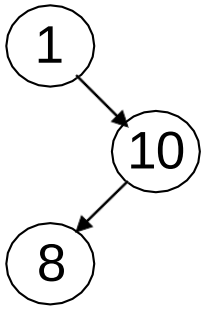
Binary Tree Insertion

1	10	8	4	6	3	2	5



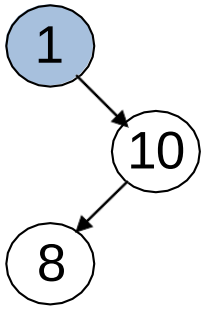
Binary Tree Insertion

1	10	8	4	6	3	2	5



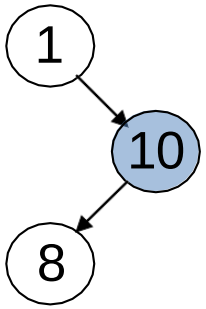
Binary Tree Insertion

1	10	8	4	6	3	2	5



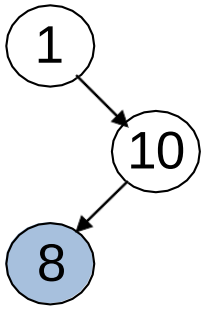
Binary Tree Insertion

1	10	8	4	6	3	2	5



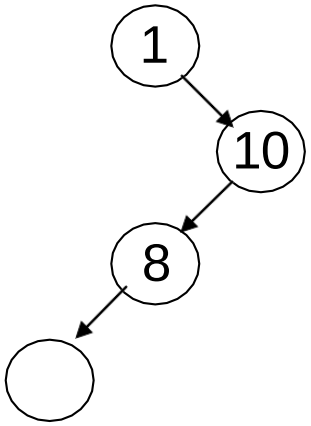
Binary Tree Insertion

1	10	8	4	6	3	2	5



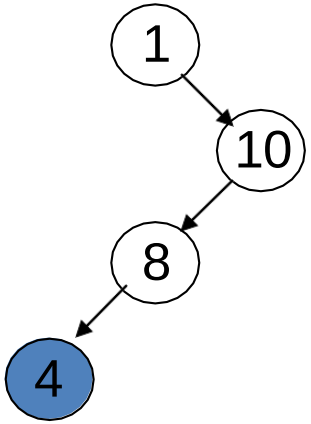
Binary Tree Insertion

1	10	8	4	6	3	2	5



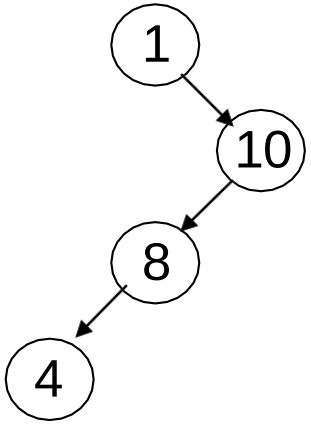
Binary Tree Insertion

1	10	8	4	6	3	2	5



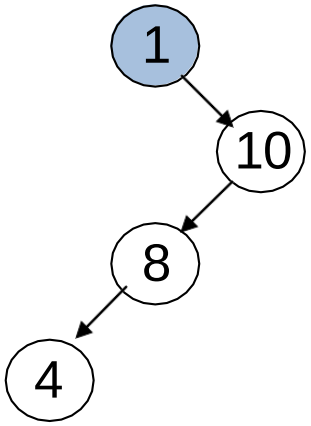
Binary Tree Insertion

1	10	8	4	6	3	2	5



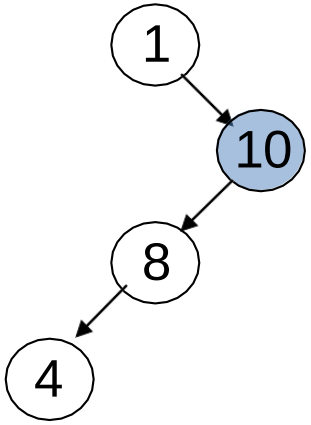
Binary Tree Insertion

1	10	8	4	6	3	2	5



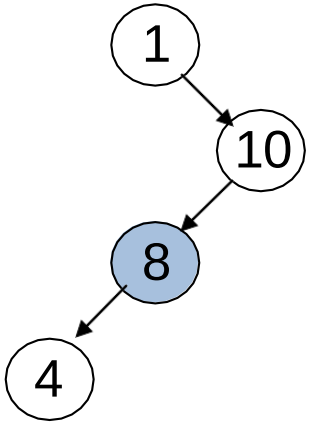
Binary Tree Insertion

1	10	8	4	6	3	2	5



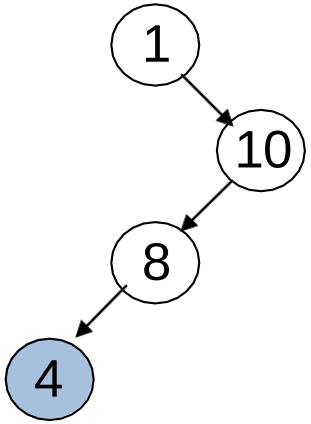
Binary Tree Insertion

1	10	8	4	6	3	2	5



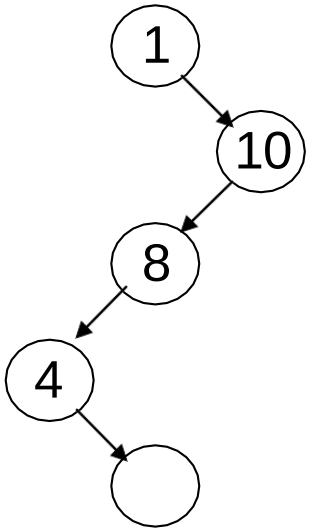
Binary Tree Insertion

1	10	8	4	6	3	2	5



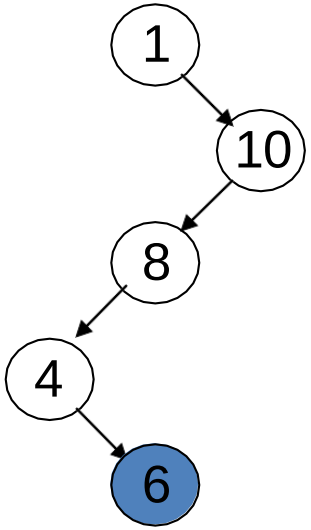
Binary Tree Insertion

1	10	8	4	6	3	2	5



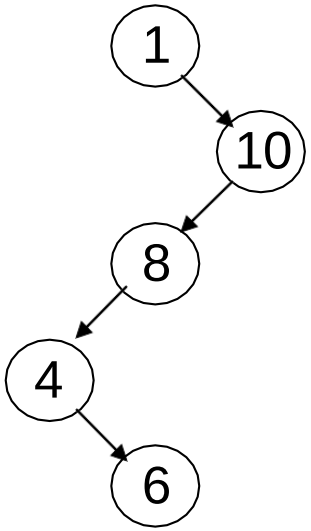
Binary Tree Insertion

1	10	8	4	6	3	2	5



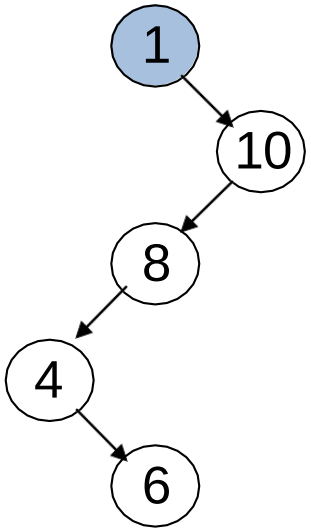
Binary Tree Insertion

1	10	8	4	6	3	2	5



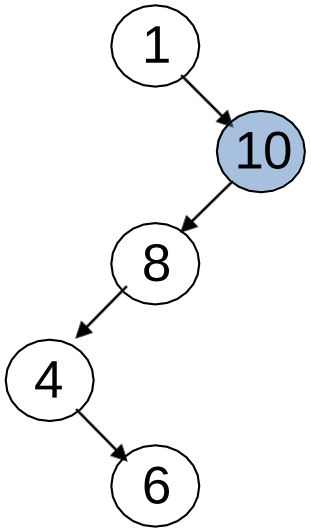
Binary Tree Insertion

1	10	8	4	6	3	2	5



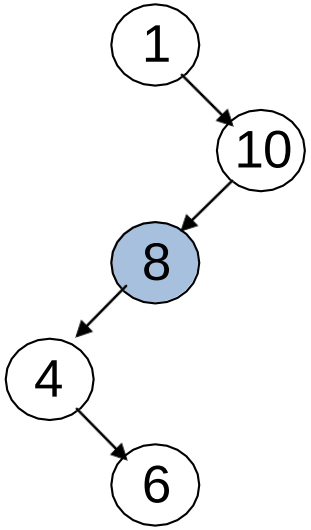
Binary Tree Insertion

1	10	8	4	6	3	2	5



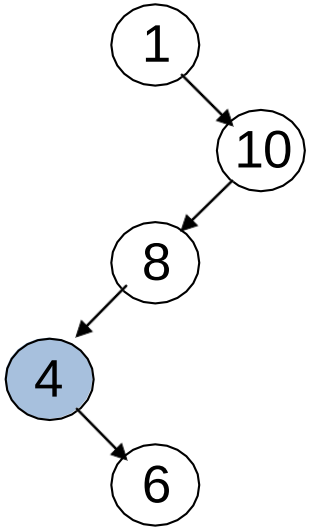
Binary Tree Insertion

1	10	8	4	6	3	2	5



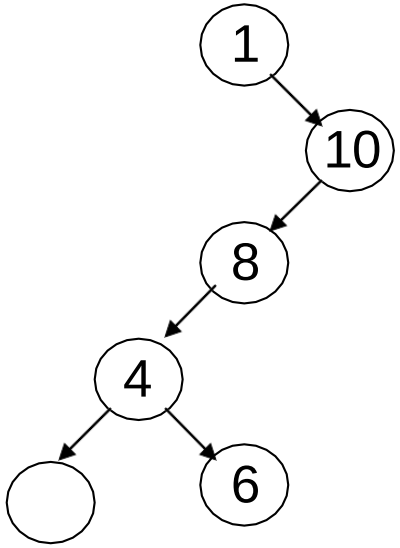
Binary Tree Insertion

1	10	8	4	6	3	2	5



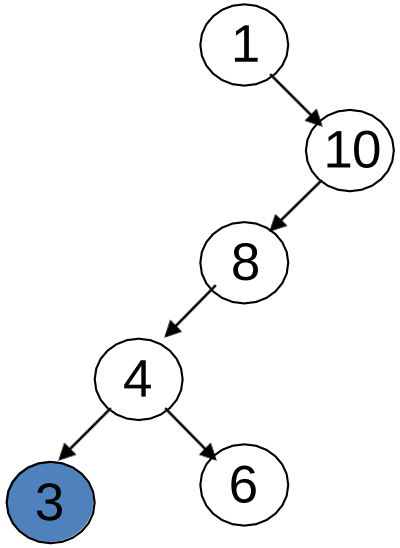
Binary Tree Insertion

1	10	8	4	6	3	2	5



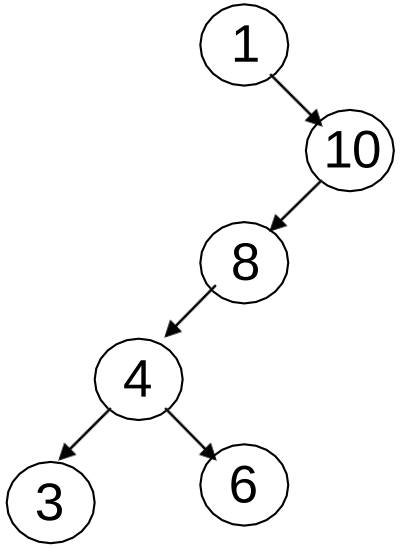
Binary Tree Insertion

1	10	8	4	6	3	2	5



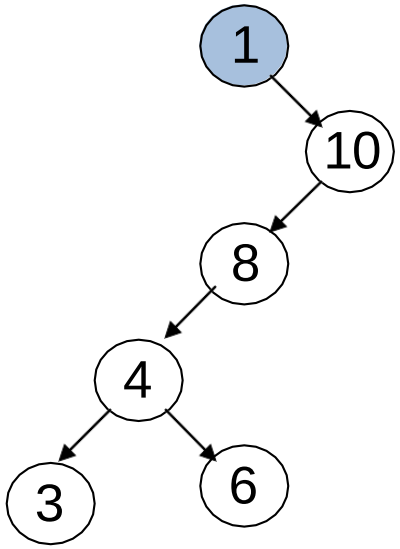
Binary Tree Insertion

1	10	8	4	6	3	2	5



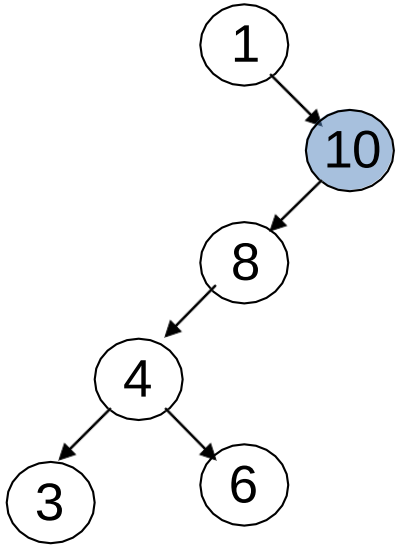
Binary Tree Insertion

1	10	8	4	6	3	2	5



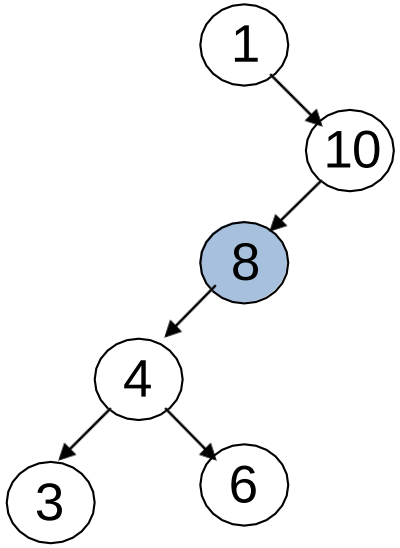
Binary Tree Insertion

1	10	8	4	6	3	2	5



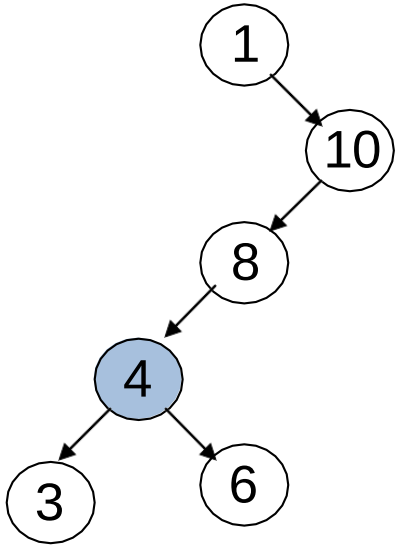
Binary Tree Insertion

1	10	8	4	6	3	2	5



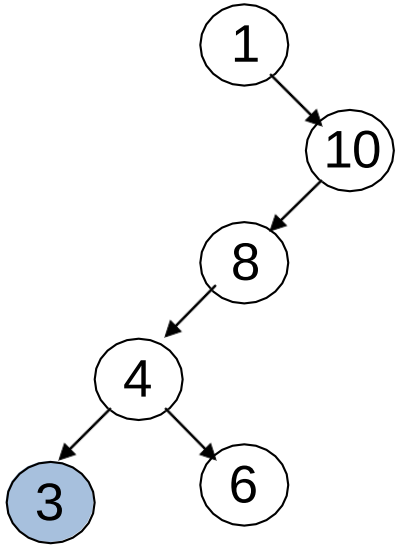
Binary Tree Insertion

1	10	8	4	6	3	2	5



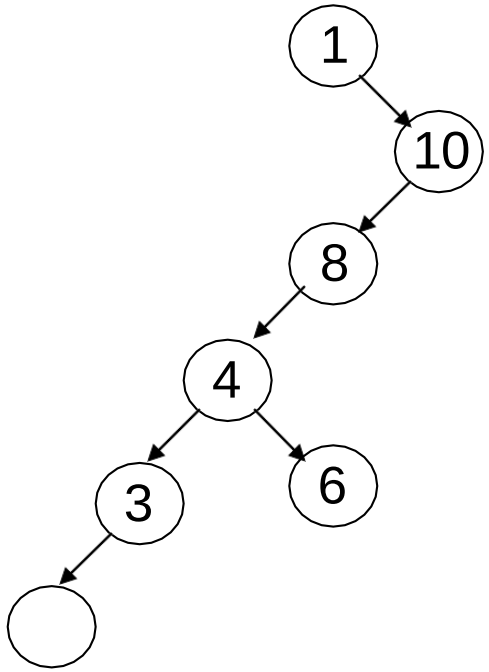
Binary Tree Insertion

1	10	8	4	6	3	2	5



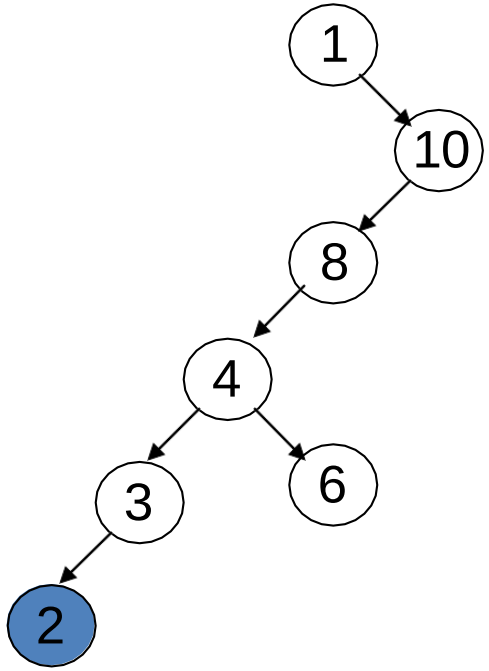
Binary Tree Insertion

1	10	8	4	6	3	2	5



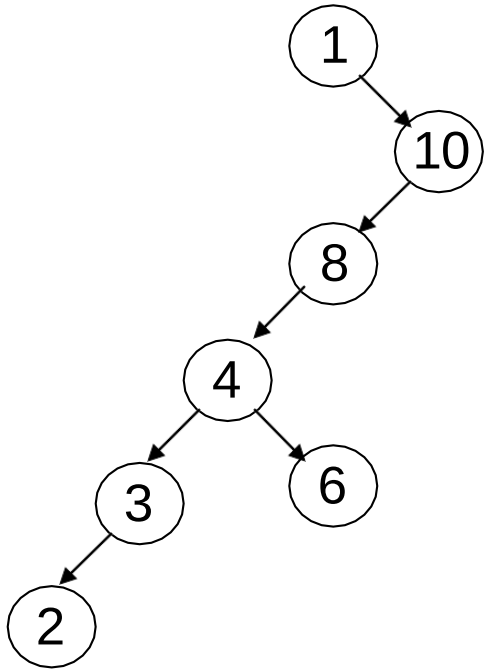
Binary Tree Insertion

1	10	8	4	6	3	2	5



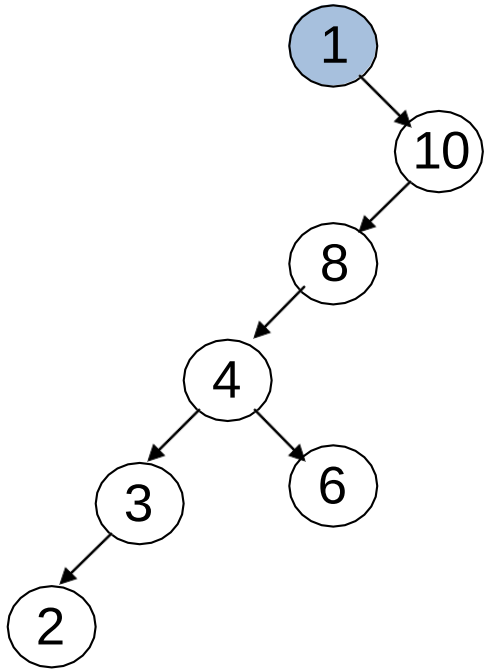
Binary Tree Insertion

1	10	8	4	6	3	2	5



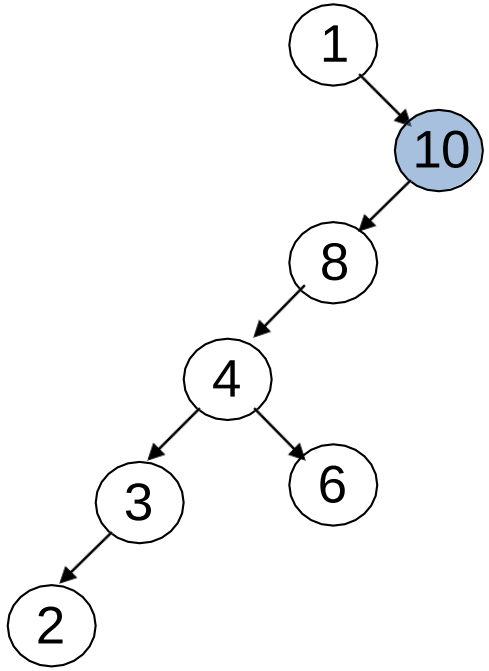
Binary Tree Insertion

1	10	8	4	6	3	2	5



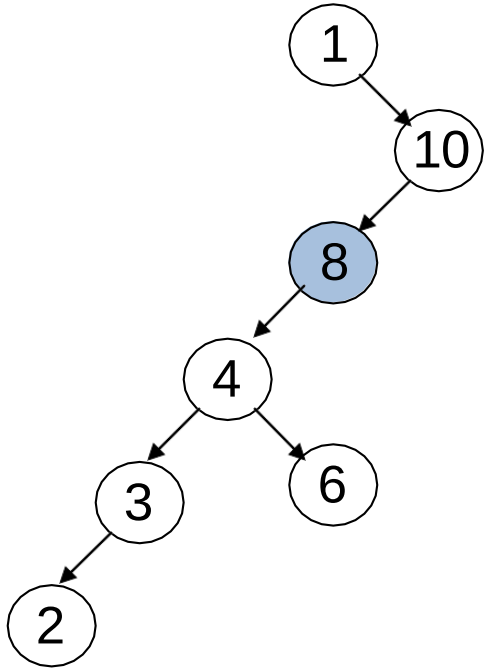
Binary Tree Insertion

1	10	8	4	6	3	2	5



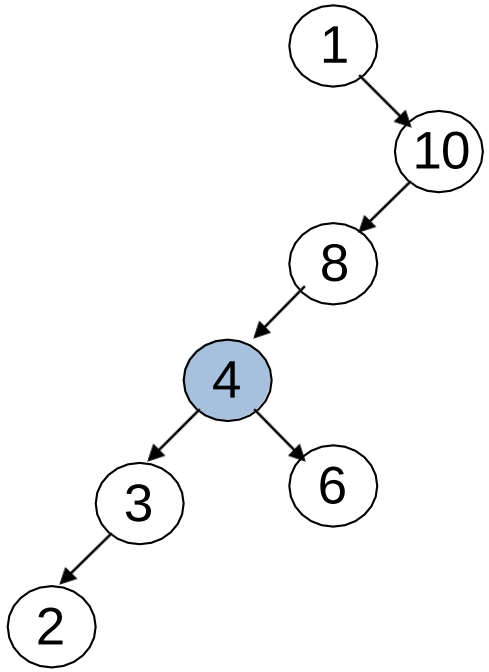
Binary Tree Insertion

1	10	8	4	6	3	2	5



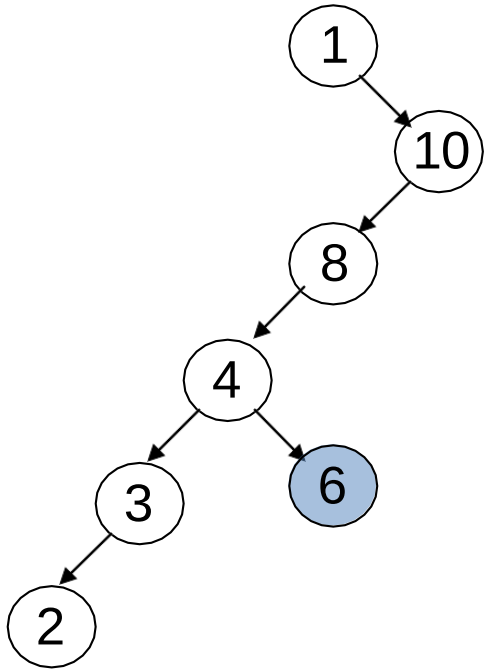
Binary Tree Insertion

1	10	8	4	6	3	2	5



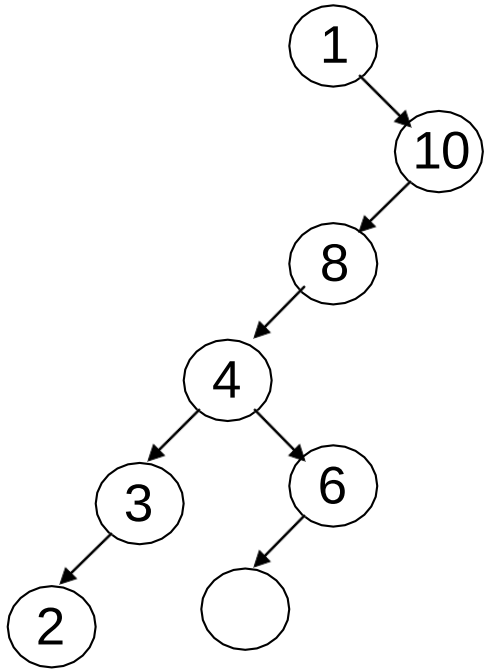
Binary Tree Insertion

1	10	8	4	6	3	2	5



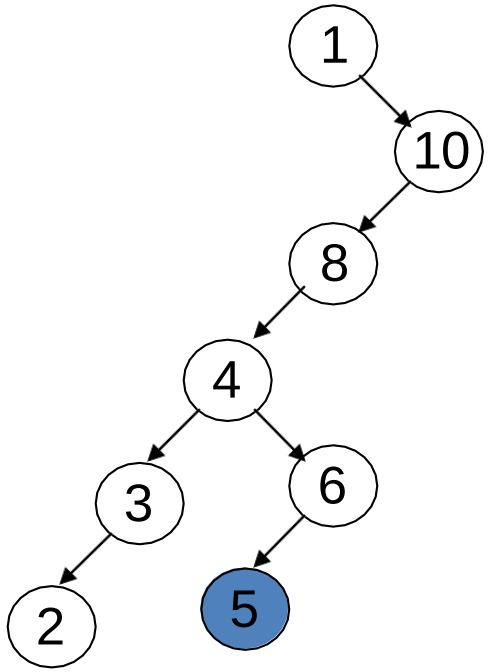
Binary Tree Insertion

1	10	8	4	6	3	2	5



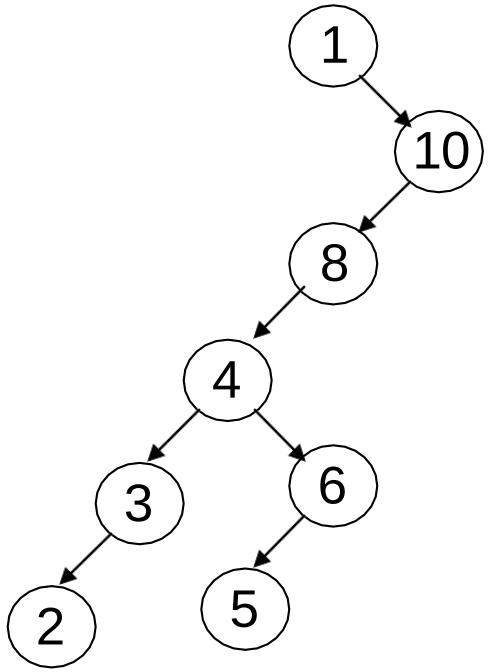
Binary Tree Insertion

1	10	8	4	6	3	2	5



Binary Tree Insertion

1	10	8	4	6	3	2	5



Deletion in Binary Tree

- We will take up three cases for how a node can be deleted from a binary search tree
  1. Deleting a Node that has No Children
  2. Deleting a Node with One Child
  3. Deleting a Node with Two Children

Deletion in Binary Tree

- **Algorithm**
  1. Perform search for value X
  2. If X is a leaf, delete X
  3. Else //we must delete internal node
     a) Replace with largest value Y on left subtree
        OR smallest value Z on right subtree
     b) Delete replacement value (Y or Z) from subtree

Note :-

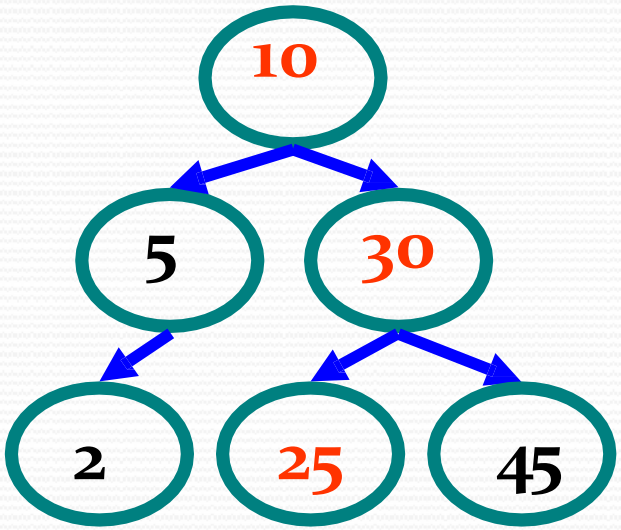
- Deletions may unbalance tree

```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
          Write "VAL not found in the tree"
        ELSE IF VAL < TREE -> DATA
          Delete(TREE->LEFT, VAL)
        ELSE IF VAL > TREE -> DATA
          Delete(TREE -> RIGHT, VAL)
        ELSE IF TREE -> LEFT AND TREE -> RIGHT
          SET TEMP = findLargestNode(TREE -> LEFT)
          SET TREE -> DATA = TEMP -> DATA
          Delete(TREE -> LEFT, TEMP -> DATA)
        ELSE
          SET TEMP = TREE
          IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
              SET TREE = NULL
          ELSE IF TREE -> LEFT != NULL
              SET TREE = TREE -> LEFT
          ELSE
              SET TREE = TREE -> RIGHT
          [END OF IF]
          FREE TEMP
        [END OF IF]
Step 2: END
```

Example Deletion (Leaf)

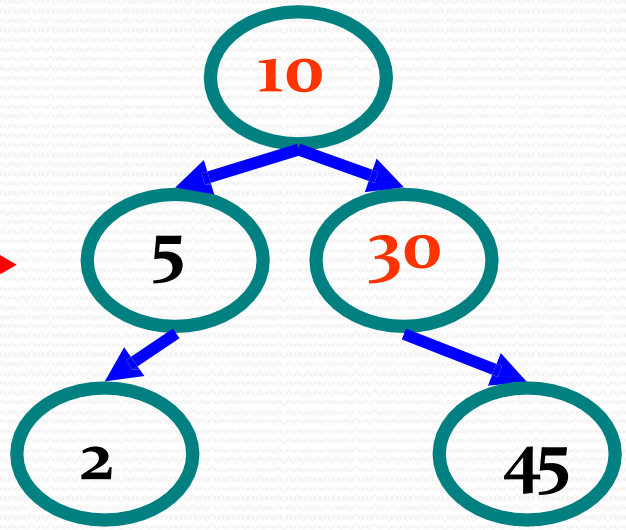
- Delete (25)



10 < 25, right

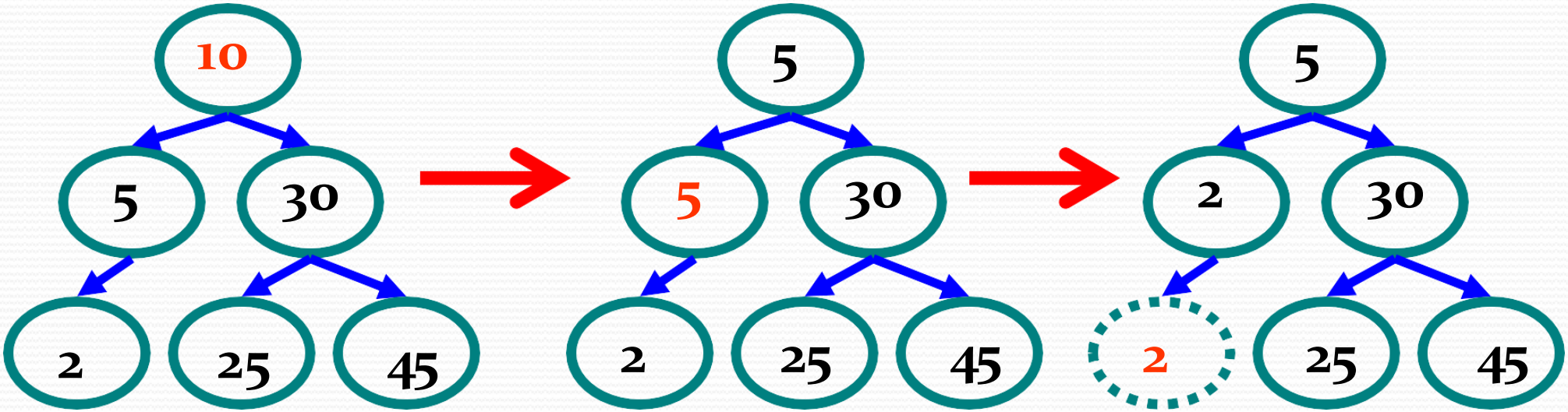
30 > 25, left

25 = 25, delete



Example Deletion (Internal Node)

- Delete (10)



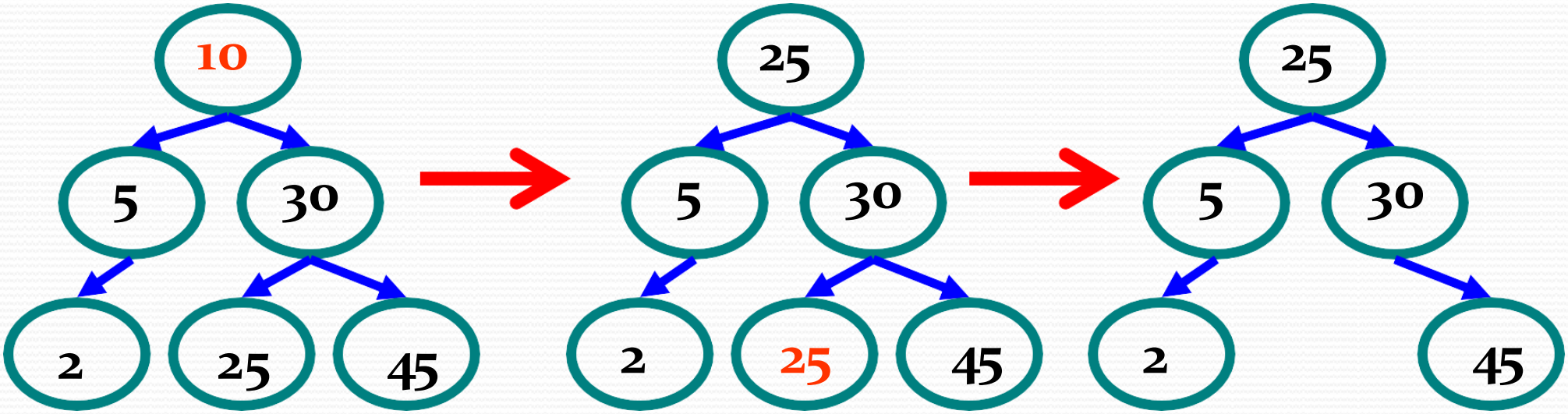
Replacing 10 with largest value in left subtree

Replacing 5 with largest value in left subtree

Deleting leaf

Example Deletion (Internal Node)

- Delete (10)



Replacing 10 with smallest value in right subtree

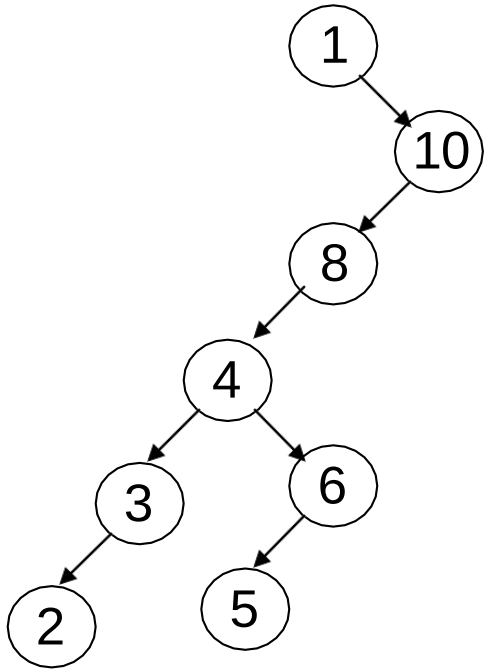
Deleting leaf

Resulting tree

Binary Tree Deletion

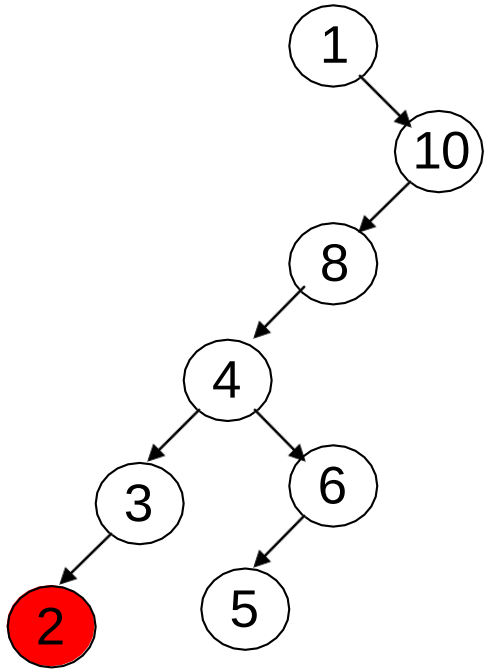
Binary Tree Deletion

1	10	8	4	6	3	2	5



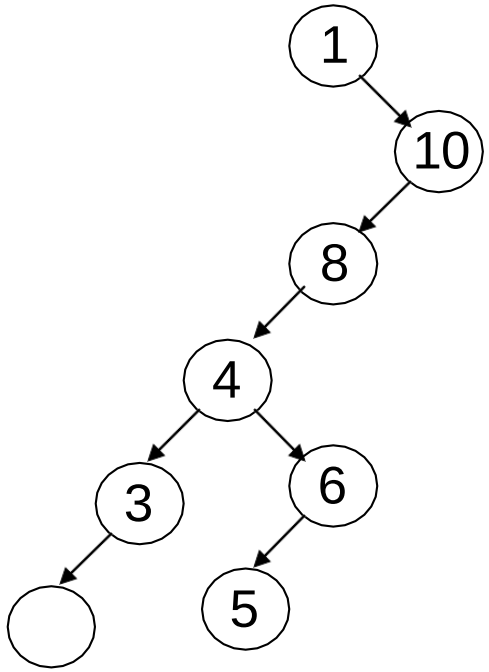
Binary Tree Deletion

1	10	8	4	6	3	2	5



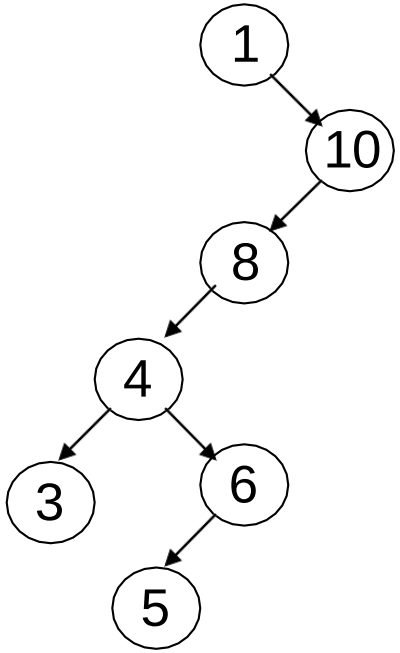
Binary Tree Deletion

1	10	8	4	6	3	2	5



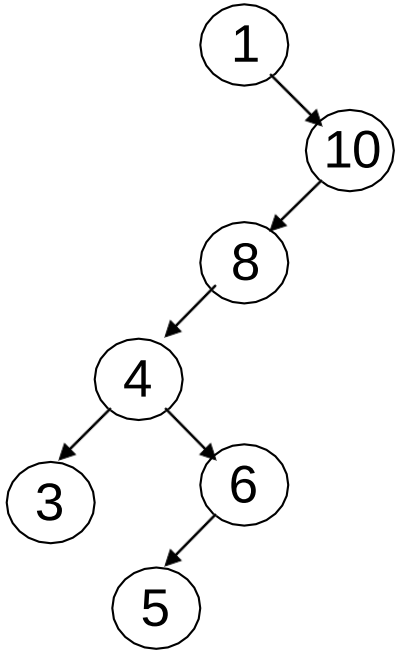
Binary Tree Deletion

1	10	8	4	6	3	2	5



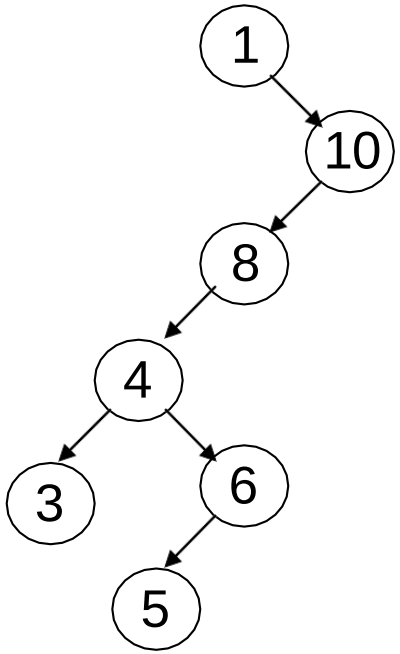
Binary Tree Deletion

1	10	8	4	6	3	5



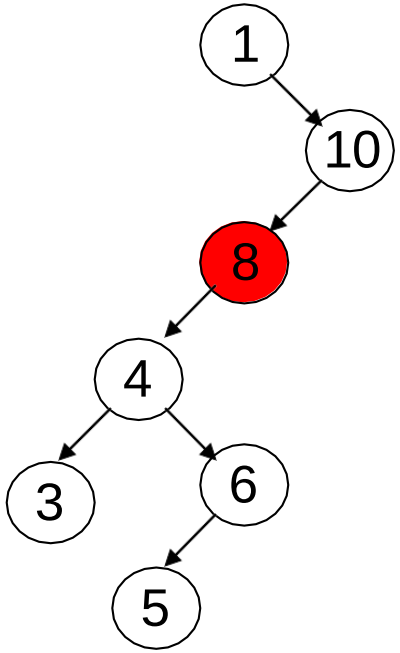
Binary Tree Deletion

1	10	8	4	6	3	5



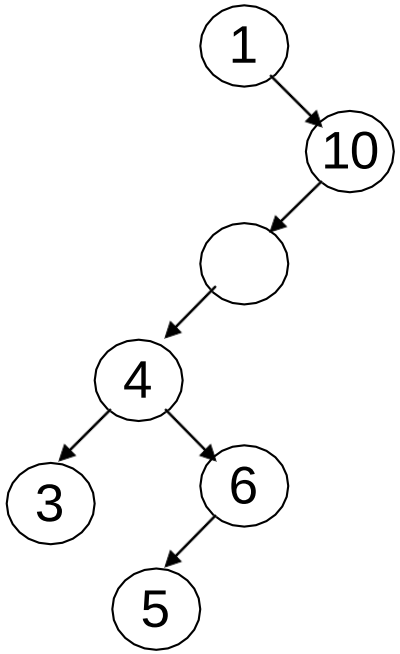
Binary Tree Deletion

1	10	8	4	6	3	5



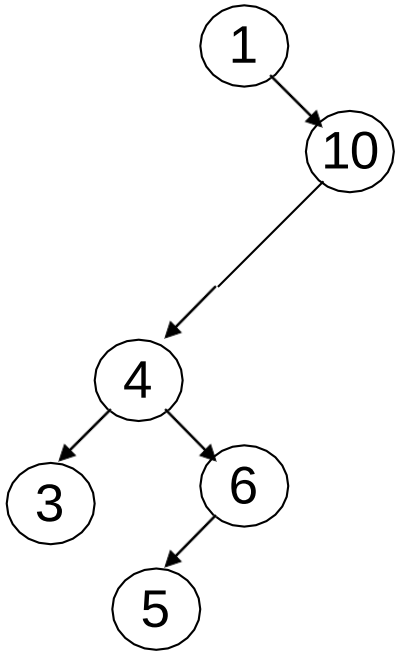
Binary Tree Deletion

1	10	8	4	6	3	5



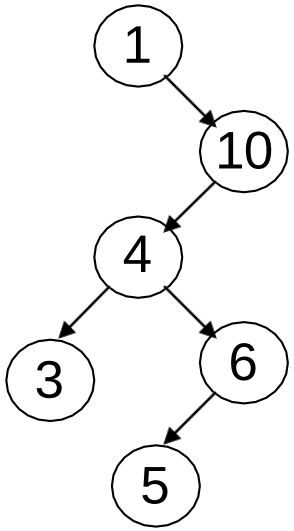
Binary Tree Deletion

1	10	8	4	6	3	5



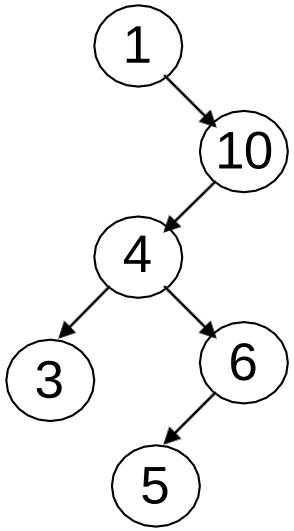
Binary Tree Deletion

1	10	8	4	6	3	5



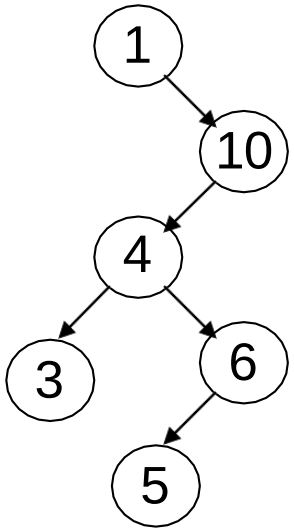
Binary Tree Deletion

1	10	4	6	3	5



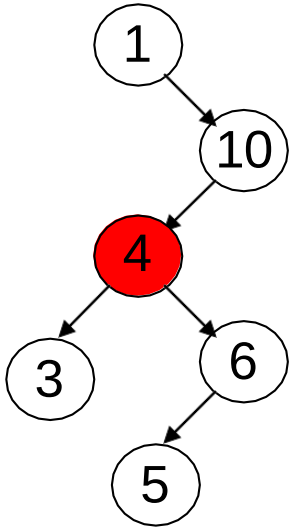
Binary Tree Deletion

1	10	4	6	3	5



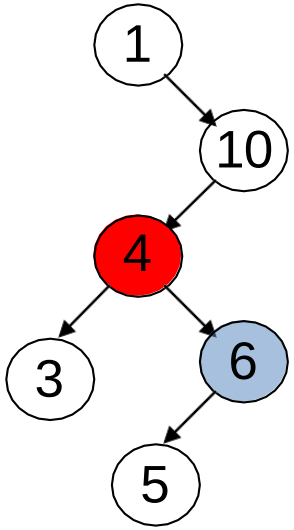
Binary Tree Deletion

1	10	4	6	3	5



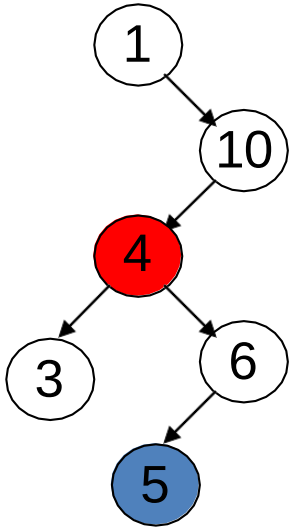
Binary Tree Deletion

1	10	4	6	3	5



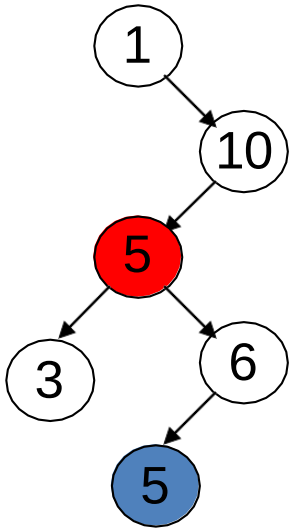
Binary Tree Deletion

1	10	4	6	3	5



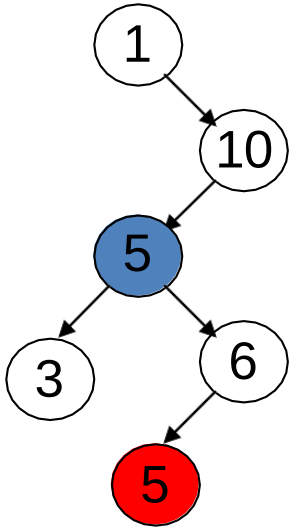
Binary Tree Deletion

1	10	4	6	3	5



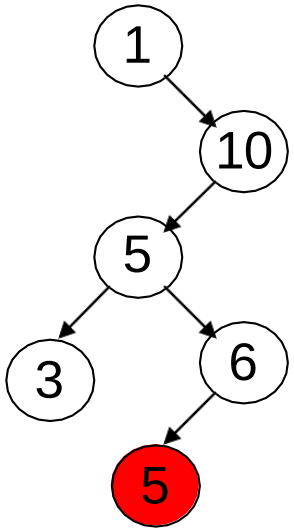
Binary Tree Deletion

1	10	4	6	3	5



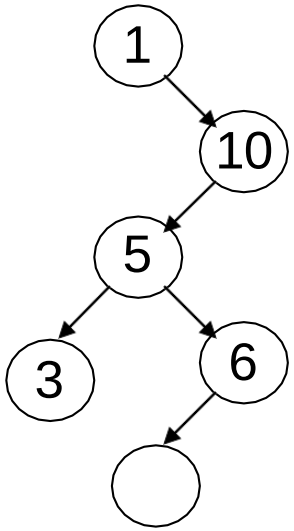
Binary Tree Deletion

1	10	4	6	3	5



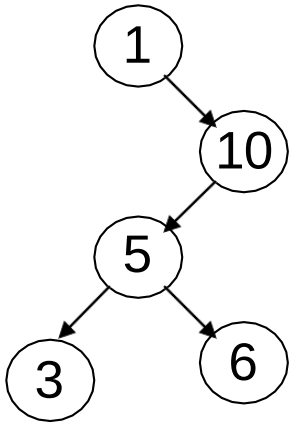
Binary Tree Deletion

1	10	4	6	3	5



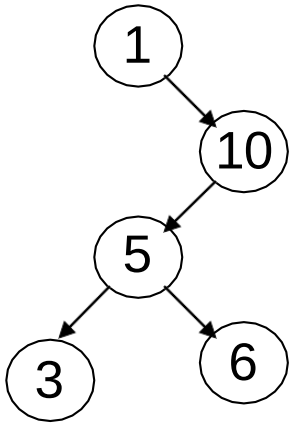
Binary Tree Deletion

1	10	4	6	3	5



Binary Tree Deletion

1	10	6	3	5

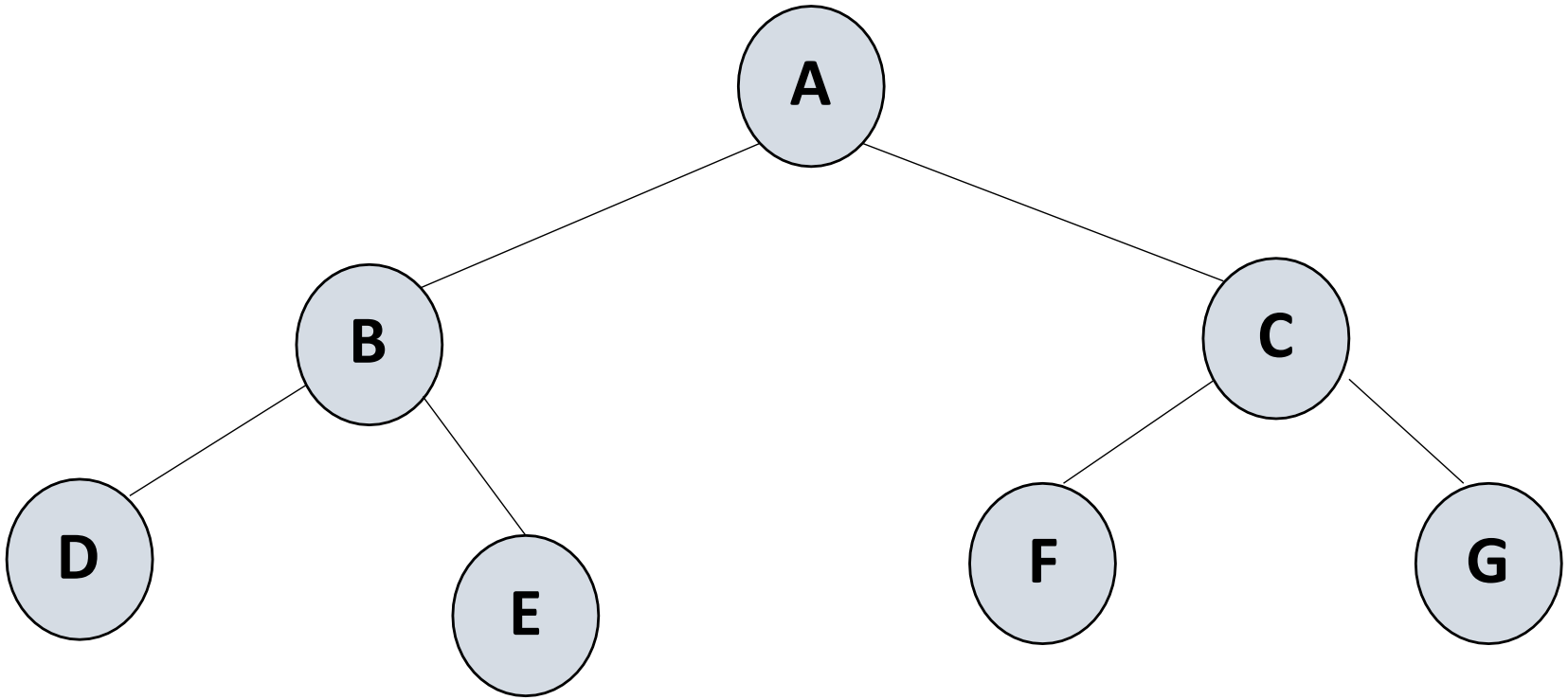


THREADED BINARY TREE

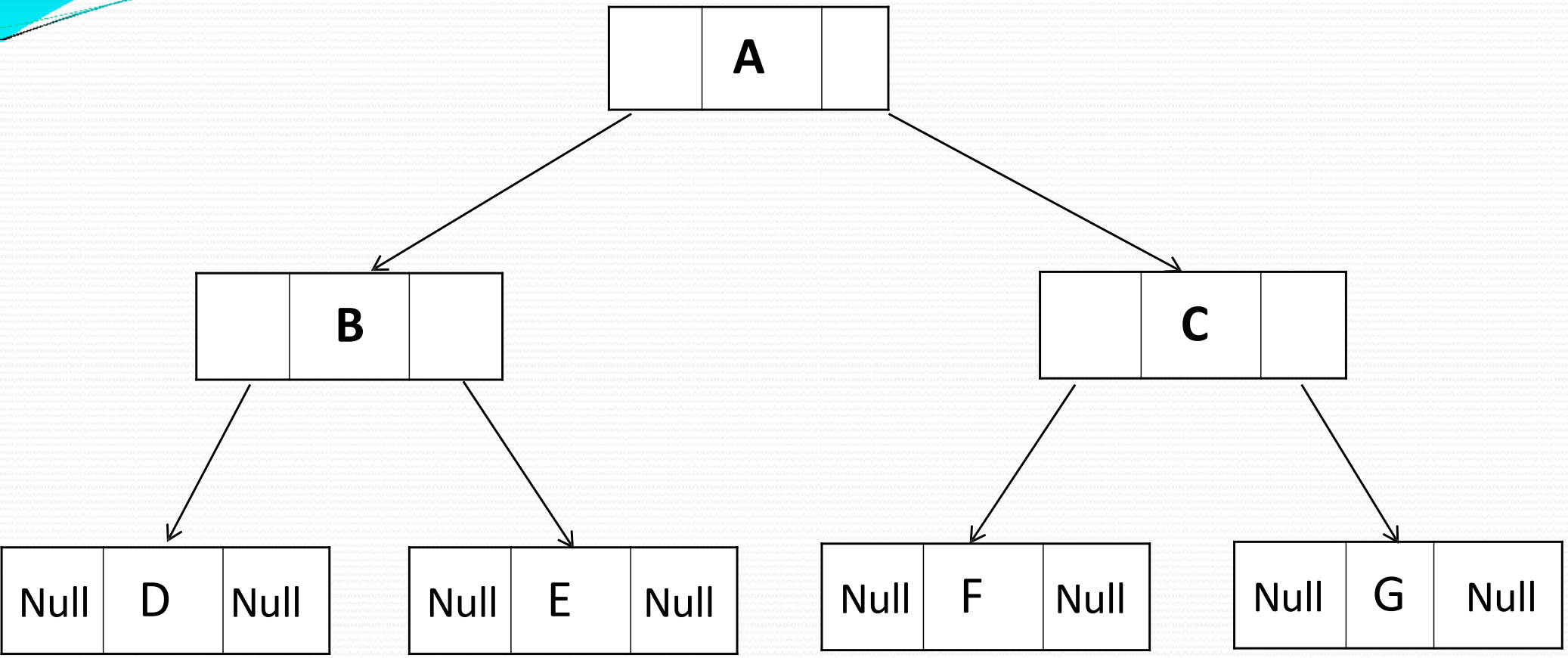
Definition:

A binary search tree in which each node uses an otherwise-empty left child link to refer to the node's in-order predecessor and an empty right child link to refer to its in-Order Successor.

A Simple Binary Tree



Threaded binary tree



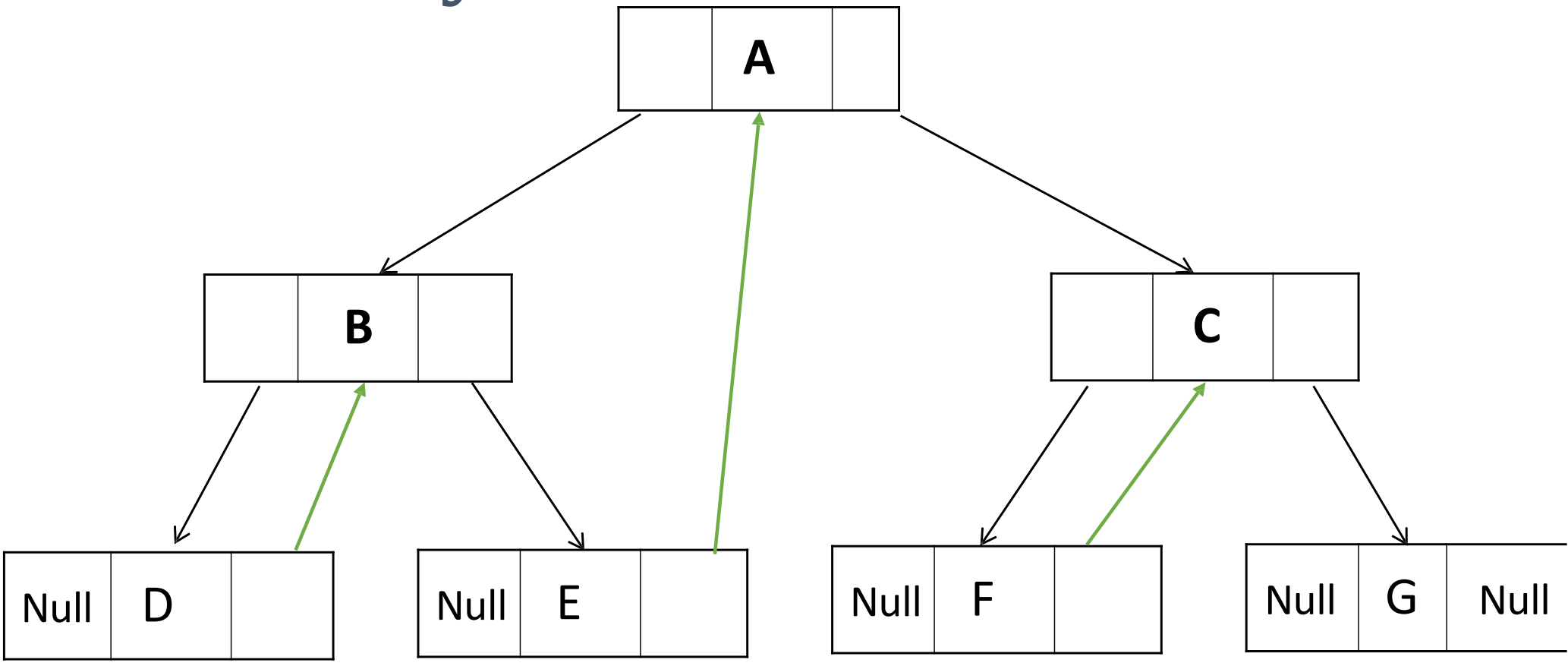
- And binary tree with such pointers are called threaded tree.
- In the memory representation of a threaded binary tree, it is necessary to distinguish between a normal pointer and a thread.

Threaded Binary Tree: One-Way

- We will use the right thread only in this case.

- To implement threads we need to use in-order successor of the tree

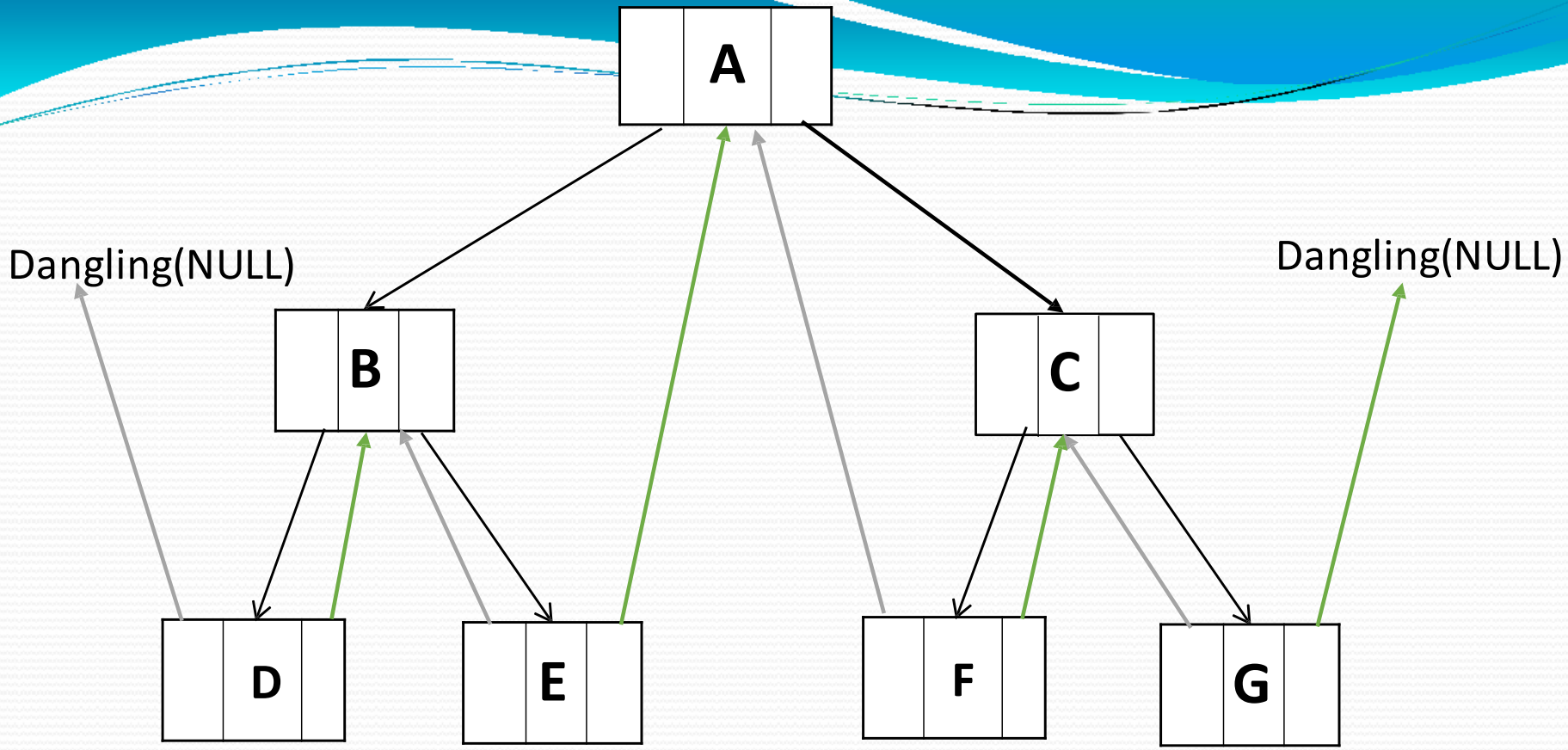
Threaded Binary Tree: One-Way



Inorder Traversal of The tree: D, B, E, A, F, C, G

Two way Threaded Tree/Double Threads

- Again two-way threading has left pointer of the first node and right pointer of the last node will contain the null value. The header nodes is called two-way threading with header node threaded binary tree.



Inorder Traversal of The tree: D, B, E, A, F, C, G

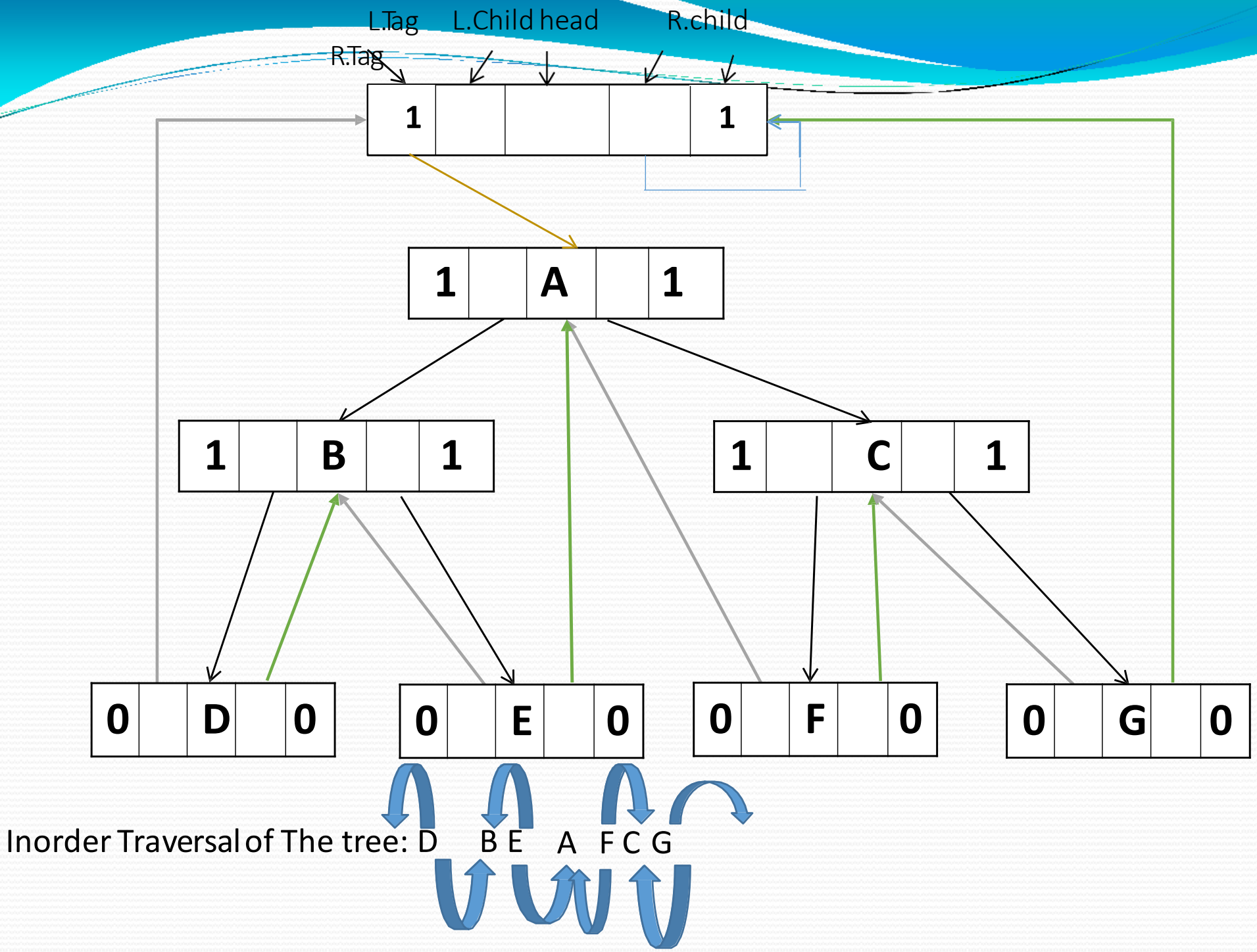
Structure of Thread BT

Node structure

For the purpose of our evaluation algorithm, we assume each node has five fields:

We define this node structure in C as:

```
Struct Node{
        int data;
    struct Node *Left,*Right;
    bool Ltag;
    boolRtag;
        };
```

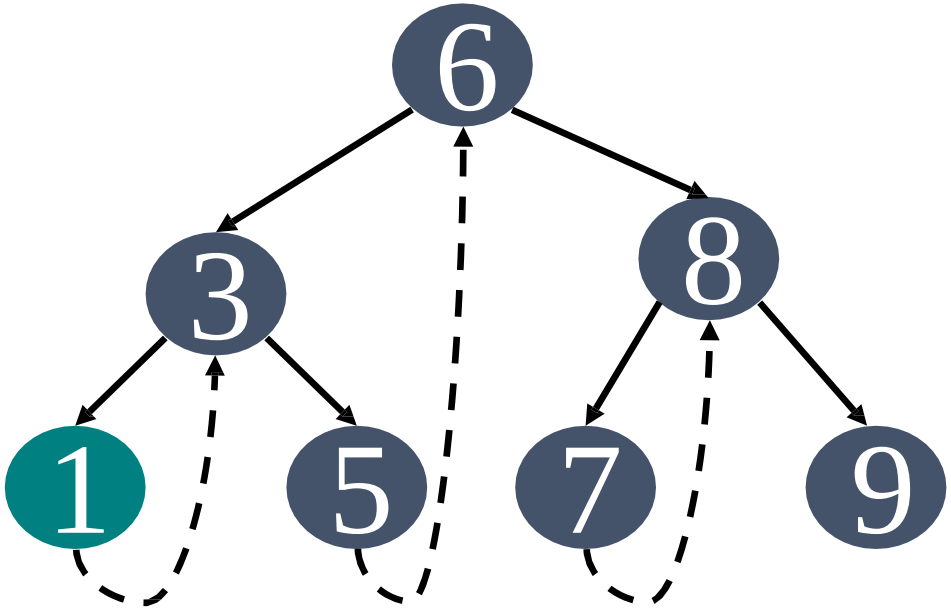


Inorder Traversal of The tree: D B E A F C G

Threaded Tree Traversal

- We start at the leftmost node in the tree, print it, and follow its right thread
- If we follow a thread to the right, we output the node and continue to its right.
- If we follow a link to the right, we go to the leftmost node,print it, and continue.

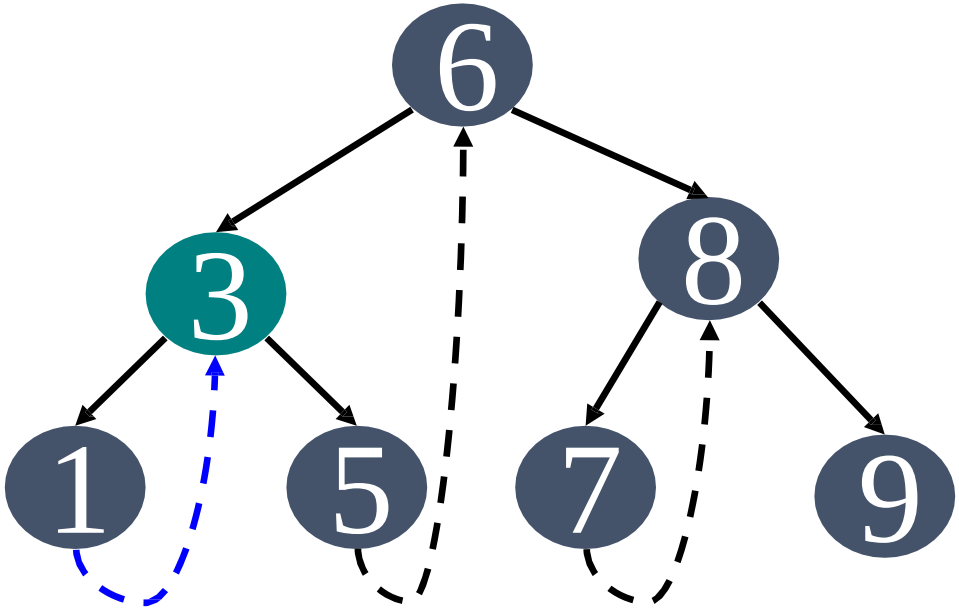
THREADED TREE TRAVERSAL



Output
1

Start at leftmost node, print it

THREADED TREE TRAVERSAL



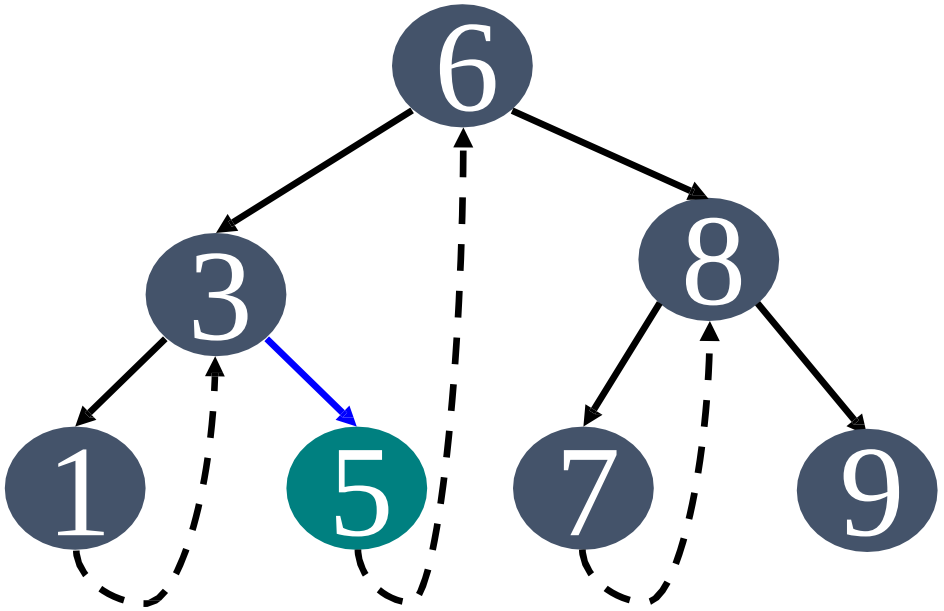
Output

1

3

Follow thread to right, print node

THREADED TREE TRAVERSAL



Output

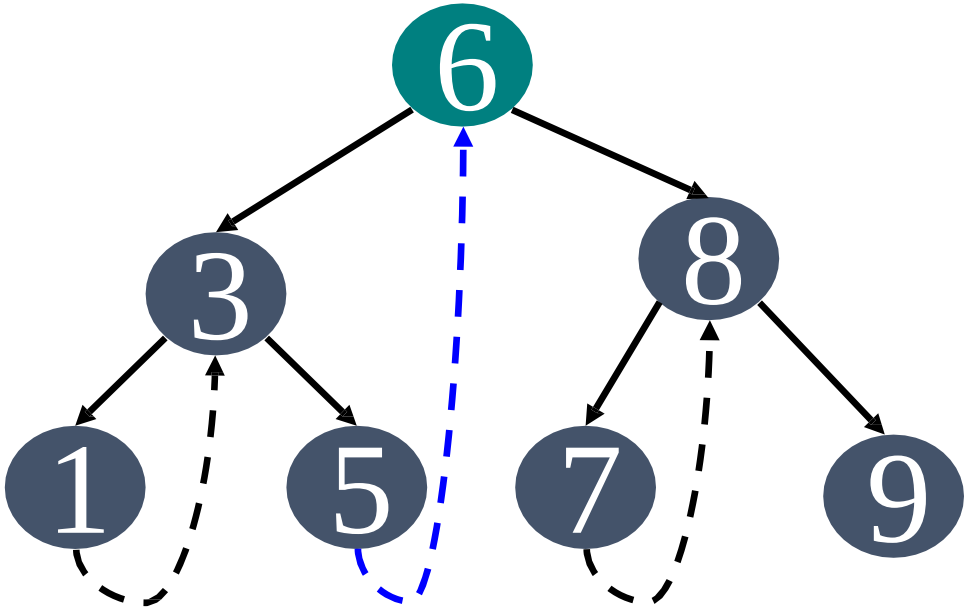
1

3

5

Follow link to right, go to leftmost node and print

THREADED TREE TRAVERSAL



Output

1

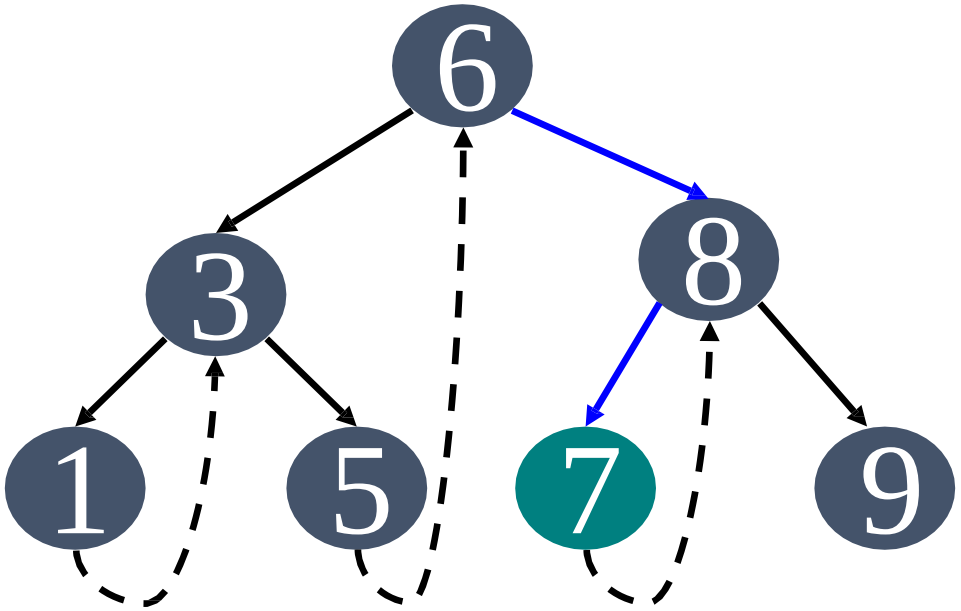
3

5

6

Follow thread to right, print node

THREADED TREE TRAVERSAL



Output

1

3

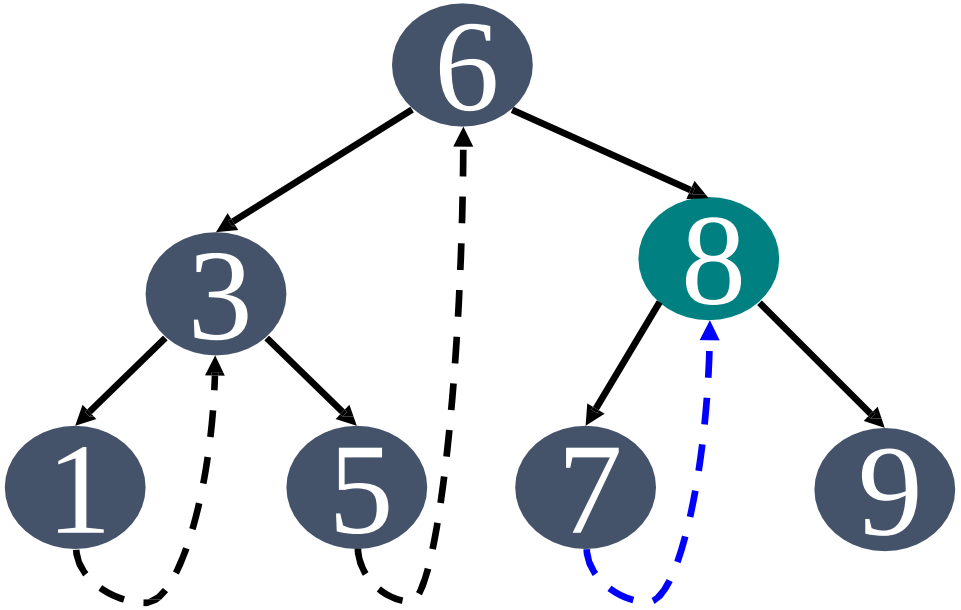
5

6

7

Follow link to right, go to leftmost node and print

THREADED TREE TRAVERSAL

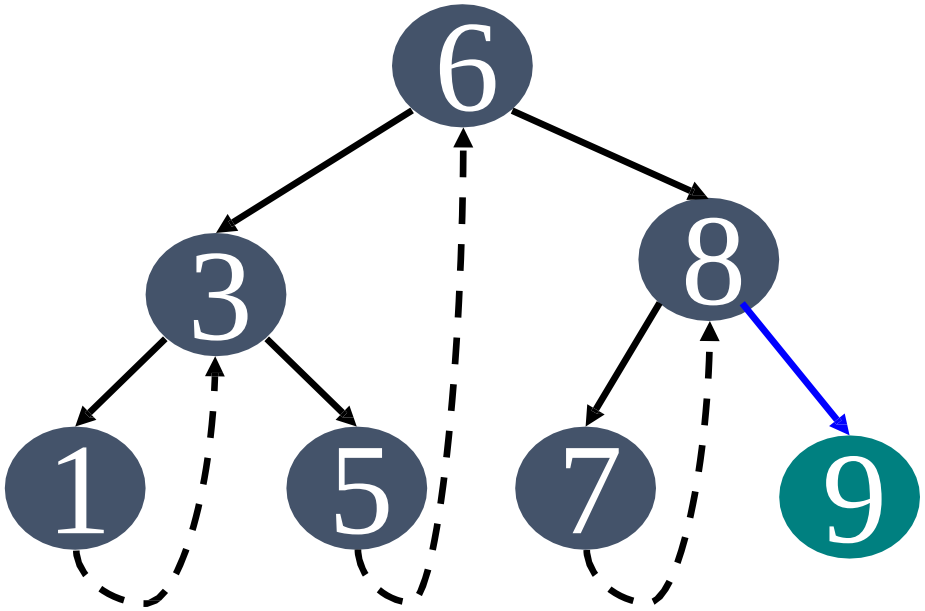


Output

1
3
5
6
7
8

Follow thread to right, print node

THREADED TREE TRAVERSAL



Output

1
3
5
6
7
8
9

Follow link to right, go to leftmost node and print

Comparison of Threaded BT

Threaded Binary Trees

- In threaded binary trees, The null pointers are used as thread.
- We can use the null pointers which is a efficient way to use computers memory.
- Traversal is easy. Completed without using stack or reccursive function.
- Structure is complex.
- Insertion and deletion takes more time.

Normal Binary Trees

- In a normal binary trees, the null pointers remains null.
- We can't use null pointers so it is a wastage of memory.
- Traverse is not easy and not memory efficient.
- Less complex than Threaded binary tree.
- Less Time consuming than Threaded Binary tree.